

LOGARITHMIC NUMBER SYSTEM
THEORY, ANALYSIS AND DESIGN

By

ATHANASIOS G. STOURAITIS

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF THE DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1986

Αφιερώνεται

στους Γονείς μου

ACKNOWLEDGMENTS

Not many graduate students are exposed to the plethora of resources and constant support that I was privileged to enjoy while pursuing my doctoral work. I take this opportunity to thank, once more, the person most responsible for this, Dr. Fred J. Taylor. He naturally became a friend in addition to advisor, being ready always to extend a helping hand and offer friendly and valuable advice.

I would also like to thank Drs. D. G. Childers, G. Logothetis, S. Y. Oh and J. T. Tou for serving on my supervisory committee.

The financial support for my studies, made available to me through an IBM educational grant, is greatly appreciated.

Special thanks are offered to Debbie Hagin for the secretarial support she constantly and unprofitably offered.

Finally, great thanks are extended to all those who work hard on the noble projects of international cooperation and peace, so needed for endeavors like this to be feasible and fruitful.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGMENTS.....	iii
LIST OF TABLES.....	vii
LIST OF FIGURES.....	viii
KEY TO SYMBOLS AND ABBREVIATIONS.....	x
ABSTRACT.....	xi
CHAPTERS	
ONE INTRODUCTION.....	1
TWO SURVEY OF LITERATURE	6
2.1 Logarithmic Conversion.....	7
2.2 Logarithmic Arithmetic.....	11
2.2.1 Multiplication and Division.....	11
2.2.2 Addition and Subtraction.....	11
2.2.3 Table Reduction Techniques for Addition/Subtraction.....	12
2.3 LNS Applications.....	14
2.3.1 Digital Filtering Using LNS.....	14
2.3.2 FFT Implementation.....	15
2.3.3 Other LNS Applications.....	16
THREE LNS AND LNS ARITHMETIC.....	18
3.1 LNS Description.....	18
3.1.1 Logarithmic Number System Representation.....	19
3.1.2 LNS Base Optimization.....	21
3.2 LNS Arithmetic Operations.....	23
3.2.1 Multiplication and Division.....	24
3.2.2 Addition and Subtraction.....	24
3.2.3 Other Arithmetic Operations.....	29
FOUR CONVERSION TO AND FROM THE LNS.....	33
4.1 Conversion From FLP To LNS.....	35
4.2 Error Analysis.....	38
4.3 Conversion From LNS To FLP	50
FIVE MEMORY TABLE REDUCTION TECHNIQUES.....	54

5.1	Essential Zeros.....	54
5.2	Optimization of Base.....	58
5.3	Other Table Reduction Techniques.....	60
SIX	ADAPTIVE RADIX PROCESSOR.....	62
6.1	Adaptive Radix Processor Principle.....	62
6.2	Error Analysis for the ARP.....	63
6.3	Presorting of Operands.....	84
6.4	Signed-digit ARP-LNS.....	85
6.4.1	Signed-digit Number Systems.....	85
6.4.2	Signed-digit Addition.....	87
6.4.3	Canonical ARP-LNS.....	91
SEVEN	ASSOCIATIVE MEMORY PROCESSOR.....	93
7.1	Associative Memory Processor Principle.....	93
7.2	Optimization.....	98
7.2.1	Latency Estimation.....	100
7.2.2	Tree Structure Experiments.....	100
7.2.3	Effect of Base on Clustering.....	103
7.3	Comparison of AMP and ARP.....	106
EIGHT	COMPLEXITY OF OPERATIONS FOR LNS PROCESSORS.....	107
8.1	Lower Bounds of Arithmetic Operations.....	107
8.2	Preconditioning of Coefficients.....	104
8.3	Simultaneous Computation of Polynomials.....	116
NINE	LNS APPLICATIONS.....	118
9.1	Multiple Input Adder.....	122
9.2	CORDIC Replacement.....	124
9.3	Pseudo Wigner-Ville Distribution.....	129
9.4	Multiplicative FIR Filters.....	135
9.4	Echo Cancellation.....	138
TEN	IMPACT OF DESIGN ON ARCHITECTURE.....	144
10.1	Two-Dimensional Memory Architectures.....	146
10.2	Three-Dimensional Memory Architectures.....	149
10.3	Fault Tolerance.....	149
10.4	Memory Assignment Schemes.....	153
ELEVEN	CONCLUSIONS.....	156
REFERENCES	160
APPENDICES		
A	DERIVATION OF TRIGONOMETRIC FORMULAS.....	170
A.1	Trigonometric Functions.....	170

A.2	Other Functions.....	172
B	PROGRAM FOR SIMULATION OF THE FLP TO LNS ENCODER.	173
C	CODE FOR GENERATION OF THEORETICAL CURVE OF THE p.d.f. OF THE LOG ENCODER ERROR. CASE i).....	177
D	CODE FOR GENERATION OF THEORETICAL CURVE OF THE p.d.f. OF THE LOG ENCODER ERROR. CASE ii).....	180
E	CODE FOR GENERATION OF THEORETICAL CURVE FOR p.d.f. OF THE LOG ENCODER ERROR. CASE iii).....	183
F	GENERATION OF APPROXIMATION CURVE FOR $f_E(E)$ (LOG ENCODER).....	186
G	CODE FOR GENERATION OF THE ESSENTIAL ZEROS ENTRIES OF TABLE 5.1.....	188
H	CODE FOR GENERATION OF THE THEORETICAL CURVE FOR THE ERROR p.d.f. OF THE LOGARITHMIC ADDER...	189
I	CODE FOR GENERATION OF THE APPROXIMATION CURVE OF THE p.d.f. OF THE ERROR AT THE OUTPUT OF THE LOGARITHMIC ADDER.....	194
J	SIMULATION OF A LOGARITHMIC ADDER DETERMINING AN EXPERIMENTAL ERROR p.d.f.	197
K	CODE USED TO GENERATE THE ENTRIES OF TABLE 6.2...	199
L	CODE FOR LATENCY OPTIMIZATION OF AMP LNS.....	205
M	CODE USED TO GENERATE THE ENTRIES OF TABLE 7.3...	211
N	PROGRAM FOR SIMULATION OF THE LNS CORDIC PROCESSOR.....	216
O	CODE SIMULATING AND ANALYZING THE FLP, FXP AND LNS VERSIONS OF ECHO CANCELLERS.....	224
	BIOGRAPHICAL SKETCH.....	232

LIST OF TABLES

TABLE	PAGE
3.1 Error Values for Several LNS Bases and for the Same Dynamic Range.....	23
3.2 Commercially Available High-Speed Memory Chips.....	28
5.1 Essential Zeros for Various Values of r and F	59
5.2 Optimal Bases for Given F and $V (=2^V)$	61
6.1 Limits of Error at the Output of the Logarithmic Adder.....	71
6.2 Experimental Values for the Error Variance for Logarithmic Addition for Two and Three-Level Partitioning and Without Using ARP at All.....	83
7.1 Ambiguous and Unambiguous States Generated by the Associative Memory Modules of the AMP.....	95
7.2 Optimal Value of $\phi_1(R)$ for AM Processor Design for Varying Values of N , K , F and $r=2$	101
7.3 Effect of Radix Variation on the Input Address Space for the AM Processor.....	105
9.1 Comparison of Commercially Available Short-Wordlength Floating-point Processors.....	119
9.2 Throughput Estimates for Various DSP Operations Using Conventional and LNS Processors.....	120
9.3 LNS Arctan-Vectoring Procedure (see Figure 9.2).....	128
9.4 Comparison of Errors in Transforming the Rectangular Coordinates to Polar Ones Using CORDIC and Through Polynomial Expansion Using FXP and LNS.....	130

LIST OF FIGURES

FIGURE	PAGE
3.1	Flowchart for LNS Multiplication/Division.....25
3.2	Flowchart for LNS Addition/Subtraction.....26
3.3	Basic LNS Arithmetic Unit.....31
3.4	LNS Hyperbolic Trigonometric Processor.....32
4.1	Hybrid Floating-point Logarithmic Processor.....34
4.2	Architecture of an FLP to LNS Converter.....37
4.3	Error Model for FLP to LNS Conversion.....39
4.4	P.d.f. for the Error in FLP to LNS Conversion. Case i).....46
4.5	P.d.f. for the Error in FLP to LNS Conversion. Case ii).....48
4.6	P.d.f. for the Error in FLP to LNS Conversion. Case iii).....51
4.7	Architecture of an LNS to FLP Converter.....52
5.1	Logarithmic Addition Mapping: $\Phi(v)$ versus v55
5.2	Logarithmic Subtraction Mapping: $\Psi(v)$ versus v56
6.1	Add Cycle Implementing the ARP Policy.....64
6.2	Error Model for Logarithmic Addition.....65
6.3	P.d.f. for $v = x - y $67
6.4	Shape of $f_E(E)$ for Case i) of LNS Addition.....77
6.5	Shape of $f_E(E)$ for Case ii) of LNS Addition.....79
6.6	Basic Computation Kernel for a Two-Transfer Addition.....89
6.7	Two-Transfer Addition Example.....90
7.1	General LNS Associative Memory Processor

	Principle.....	97
7.2	Tree-Structured AM Module of AMP (see Table 7.1).....	102
7.3	Number of Switches versus F for Tree-Structured AM Modules.....	104
9.1	Three-Operand LNS Adder.....	123
9.2	LNS CORDIC Architecture.....	127
9.3	LNS Systolic Wigner Processor Architecture.....	134
9.4	LNS-MFIR Architecture.....	137
9.5	General Model of Echo Canceller.....	140
9.6	Convergence Curves for Three Echo Canceller Implementations: a) Floating-point, b) Fixed-point and c) LNS.....	142
10.1	C ⁵ Architecture.....	145
10.2	Dance Hall Architecture.....	147
10.3	3-D C ⁵ ARP LNS Architecture.....	150
10.4	Intraboard Communication of 3-D Architectures via Cell Interfaces.....	151

KEY TO SYMBOLS AND ABBREVIATIONS

<u>Symbol</u>	<u>Explanation</u>
LNS	Logarithmic Number System
FLP	FLoating-Point number system
FXP	FiXed-Point number system
r	Base of the LNS
V	Dynamic range of the arithmetic system
I	Number of integer bits
F	Number of fractional bits
$\log x$	Natural logarithm of the nonnegative number x
$\log_r x$	Logarithm to the base r of x
$\log_2 x$	Logarithm to the base 2 of x
p.d.f.	Probability Density Function
$U[-a, a]$	Uniform p.d.f. over $[-a, a]$
$E[x]$	Expected value of x
$RND[x]$	Round x to the closest acceptable discrete value
$\lfloor x \rfloor$	Floor of x : largest discrete value less than or equal to x
$\lceil x \rceil$	Ceiling of x : smallest discrete value larger than or equal to x
DSP	Digital Signal Processing
ARP	Adaptive Radix Processor
CAM	Content Addressable Memories
AM	Associative Memories
AMP	Associative Memory Processor
SD	Signed-Digit Number System
$f_E(E)$	p.d.f. for the random variable E
Z_M	Residue class modulo M
$(FU)^2$	Florida University Floating-point Unit

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

LOGARITHMIC NUMBER SYSTEM
THEORY, ANALYSIS AND DESIGN

By

ATHANASIOS G. STOURAITIS

August 1986

Chairman: Dr. Fred J. Taylor
Major Department: Electrical Engineering

The logarithmic number system (LNS) has recently been shown to offer a viable alternative to traditional computer arithmetic number systems, when a large dynamic range and high precision are required. This dissertation can be considered as part of a recent attempt of the scientific community to further investigate and enhance the qualities of LNS. It broadens the spectrum of available LNS arithmetic operations, optimizes its design parameters, enhances memory reduction techniques and alleviates problems related to the short wordlength of currently existing high-speed memory tables. Two techniques, based on sequential and random memory access principles, are investigated, through the vehicles of stochastic analysis and dynamic programming, with exciting results, at least for one of the designs. The

impact of this design on the processor architecture is analyzed and its consequences on better handling the issues of fault tolerance and recovery through reconfiguration are examined. An attempt to systematize the development of applications in order to fully exploit the LNS attributes is made and the theoretical operations count bound is approached. Finally a series of Digital Signal Processing and other applications, where the conventional systems present drawbacks, is considered and appealing LNS solutions are offered. It can be claimed that the results of this research make feasible the VLSI realization of a 20-bit LNS processor.

CHAPTER ONE INTRODUCTION

Using traditional or modified von Neumann architectures, attached array processors or Digital Signal Processing (DSP) chips, DSP has undergone a decade of rapid development and enjoyed many successes. DSP scientists and technologists have been accustomed, over this period, to responding to the requirements for increased performance. However, there is a list of problems (multidimensional or real time), which often require processing power that the current generation machinery can only provide "off-line." Other problems, being adaptive, require a dynamic reconfiguration of the system during run-time. Others may require multitasking, which precludes the use of highly tuned unitask DSP architectures (e.g. systolic arrays). Other issues of major importance are the ones of fault tolerance and recovery. The above list of advanced requirements creates an imperative need for faster and more intelligent and accurate systems.

Recent developments in the area of the Logarithmic Number System (LNS) have proven it to offer some significant advantages, when compared to other number systems. The conventional weighted binary systems, being

either all-fractional, all-integer or floating-point (FLP) systems present the problems of relative slowness or high circuit complexity for operations like multiplication or division. On the other hand, modular number systems, like the Residue Number System (RNS), which is very fast due to lack of necessity for propagation of carries, face difficulties, when dealing with the operations of division, overflow detection and magnitude comparison. The main advantages of LNS are the extended dynamic range on which it operates and the high speed of operations, accompanied by a remarkably regular data flow. The LNS though, while exhibiting many positive attributes, has not enjoyed the attention given to other systems. Therefore, a number of fundamental issues remain unsolved at this time. For the LNS to be fast enough to justify its use, it has to be memory intensive, whenever it deals with add/subtract operations. The high-speed memory chips, though, are technologically restricted to a limited wordlength of ~12 bits. There is a need for methods allowing to increase the wordlength, for the LNS to be sufficiently accurate. This is one of the tasks accomplished in this dissertation. Two advanced LNS processors are examined. These are the Adaptive Radix Processor (ARP) and the Associative Memory Processor (AMP). The former, partitioning the dynamic range, is a "random memory access" approach to the problem of enhancing the precision of LNS processors, while the

latter is a "sequential access" one. The two processors are researched and analyzed. A statistical analysis on the error budgets was performed using stochastic analysis mechanics and dynamic programming techniques. The addressing space was optimized using table reduction techniques. The two processors were compared with regard to latency and the total number of storage-bits required in order to obtain the same precision. The sensitivity of AMP to the location of the radix is also investigated. Alternative number systems were considered for integration with the best of the two designs (ARP) to achieve even better results.

Floating-point number representation is the dominating choice of systems designers, whenever extended dynamic range and high precision are simultaneously required. Due to the importance of this system, a full-scale statistical analysis of the error involved in the conversion from FLP to LNS was conducted.

Since one of the major advantages of LNS over other number systems is the increased speed at which multiplications and divisions are performed, the lowest bound of algorithms for addition and subtraction was targeted. The current form of useful DSP operations, like FFTs, convolutions, correlations, etc., was checked against these theoretical bounds.

Specific applications of LNS were also examined, with an effort to systematize the whole procedure for

development of applications on a special-purpose LNS processor, exploiting the findings of the present research for optimal outcomes. Some other applications with unique and appealing LNS solutions were considered as well.

The architectural consequences of the new processor designs were also investigated. The regularity of the data flow enabled to envision 3-D systems, where besides communicating in a traditional plane (card level), multiple and regular communication paths are opened in the third dimension (intracard level).

The successful accomplishment of the above research tasks resulted in a fully theoretically explored and experimentally tested LNS processor, capable of operating at the speed of $10^8 - 10^9$ floating-point like operations per second (FLOPS) over an extended dynamic range and with sufficient precision to serve as a special purpose processor in a number of applications, where real-time processing is required. A highly desirable processor evolved in terms of throughput, cost, complexity and flexibility.

The organization of the dissertation is as follows: Chapter Two contains a comprehensive survey of the existing literature. This survey is organized along functional lines and comprises broadly a description of LNS arithmetic and applications related literature. Chapter Three offers the LNS number representation followed throughout this work, along with already existing

or developed here algorithms for basic arithmetic operations. Chapter Four deals with the conversion problem and Chapter Five summarizes existing memory reduction techniques and suggests new ones along with memory optimization procedures. Chapter Six describes the ARP and analyzes its statistics. It also suggests alternative hybrid Signed-digit LNS designs. In Chapter Seven the AMP design is analyzed for optimal results. Existing theoretical bounds of operation count for basic DSP algorithms are offered in Chapter Eight, and several techniques for optimizing the multiplication/addition count tradeoff for LNS are offered. In Chapter Nine a systematic application development procedure is offered. It is accompanied by applications of LNS in areas, where conventional systems present drawbacks, or the specific designs present interesting features. The impact of the dominating LNS design (ARP) on the architecture and the issues of fault tolerance and recovery is investigated in Chapter Ten. In Chapter Eleven, by way of summarizing the proceedings, the significance of the described research is brought out, and some directions for future research are indicated. Finally, some of the arithmetic algorithms are derived in Appendix A, and lists of the C-language programs used to simulate and verify experimentally various theoretical conjectures are given in the rest of the Appendices (B - O).

CHAPTER TWO SURVEY OF LITERATURE

As is usually the case with basic research, theory for Logarithmic Number System(s) (LNS) remained virtually unused until the advances in semiconductor memory technology offered the vehicle for actual implementation of sufficiently efficient LNS engines. Work related to LNS (mainly conversion routines) dates back to Briggs three centuries ago [Sal54]. In 1971 Kingsbury and Rayner [Kin71] first outlined logic hardware and software function approximations to the addition and subtraction of two positive numbers logarithmically encoded. Swartzlander and Alexopoulos [Swa75] followed with an analysis of ROM-based hardware for faster addition but with a wordlength limitation of 12 bits. Later Lee and Edgar [Lee77a-b] developed computationally efficient supporting algorithms for microcomputer control and signal processing applications to be implemented using 8 and 16-bit logarithmic arithmetic on microprocessors. They established that unlike floating-point arithmetic, which provides accuracy at the expense of speed, the LNS provides for both and is suitable for Digital Signal Processing (DSP) applications. Kurokawa et al. [Kur80] applied LNS to the implementation of digital filters and

demonstrated that it gives filtering performance, superior to that of a floating-point system of equivalent wordlength and range. Similar observation was made by Swartzlander et al. [Swa83] when using LNS for a Fast Fourier Transform processor. Shenoy [She83] applied LNS to Least Mean Squared Adaptive digital filters, to achieve increased performance, when compared to fixed-point realizations. These studies showed that all the basic arithmetic operations are very fast and easy to implement in LNS. The technological advances in memory construction have renewed interest in the LNS research over the last few years. A brief survey of the literature in the LNS will now be presented.

2.1 Logarithmic Conversion

Cantor et al. [Can62] have described a special-purpose structure to implement sequential table look-up (STL) algorithms for the evaluation of logarithms. Tables of precomputed constants are used to transform the argument into a range, where the function may be approximated by a simple polynomial. Their work was based on previously developed transformation algorithms by Bemmer [Bem58]. Later, Specker [Spe65] offered alternative STL algorithms with a smaller number of constants required, which could either be wired into a computer structure or programmed. For an R -bit word-format, a basic hardware configuration required R additions and $R/2$ shift

operations. Mitchell [Mit62] proposed a method of computer multiplication and division using binary logarithms and performed an analysis to determine the maximum errors that may occur as a result of the approximation used. He determined the binary logarithm of a binary number in hardware from the number itself, by using simple logical and shift operations after encoding the binary number into a form, from which the characteristic is easily determined and the mantissa easily approximated. This work was later expanded by Combet et al. [Com65]. They partitioned the range of the binary numbers into four parts and again made a piecewise linear approximation. The linear equations, which they used, were found by trial and error, using a criterion of minimum error, and constraining the coefficients to be easily implemented with binary circuitry. That is the coefficients were chosen to be fractions with integer numerators and power of two denominators. With a four-subinterval partition, the single division error was reduced by a factor of six. In Hall et al. [Hal70] the authors discuss algorithms for computing approximate binary logarithms, antilogarithms and applications to digital filtering computations. They define the coefficients for a linear least square fit of the binary logarithm of the fractional number incremented by one, partitioning again the range of numbers into four subintervals. The maximum error with these coefficients is reduced by a factor of roughly 1.3, but two additional

sums are required. Marino [Mar72] presents a parabolic approximation method for generation of binary logarithms, with a precision increased over Hall et al. [Hal70] by a factor of 2.5, without increasing the number of sums. Their method can be implemented as a subroutine or using a hardware peripheral device to build, with almost the same precision, an approximate generation of many useful elementary functions. Kingsbury and Rayner [Kin71] propose the use of linear analog to digital and digital to analog converters in the case of digital filtering employing LNS. Conversion of digital logarithms to analog voltages can be achieved by supplying an amplifier chain from a reference voltage and causing each bit of the logarithm to switch the overall gain by a factor depending on the significance of the bit. A/D conversion is achieved by using a high-gain comparator with the D/A converter to approximate successively the converter output to the signal input. The use of an interesting RC circuit to implement the logarithmic Analog to Digital conversion is proposed by Duke [Duk71]. Chen [Che72] presents unified methods for evaluation of exponentials, logarithms, ratios and square roots for fractional arguments, in one conventional multiply time, employing only shifts, adds, high-speed table look-ups and bit counting. Briggs used decimal digit by digit schemes [Sal54] for evaluating logarithms. Meggitt's [Meg62] unified approach was based on Briggs' scheme, which was further improved by Sarkar and

Krishnamurthy [Sar71]. Walther [Wal71] showed another unifying algorithm containing the CORDIC schemes proposed by Volder [Vol59]. The elementary functions he dealt with are inferred from hyperbolic functions, invoking repeatedly a basic shift sequence. A method suggested by de Luggish [Lug70] can be quite advantageous, when the add time is the cost overriding factor. Very recent methods for logarithm generation include the sequential squaring method by Karp [Kar84], useful for large scale vector processors and limited precision microprocessors, and the Difference Grouping Programmable Array Logic (DGPLA) method discussed by Lo and Aoki [Lo85]. The latter is based on the work of Brubaker and Becker [Bru75], which utilizes read-only memories (ROMs) for the generation of the logarithm. For a predetermined error, the number of bits can be chosen for transformations; otherwise, for a predetermined number of bits, the error can be set to an optimal minimum value by the adjustment between the upper and lower transformations. The primary drawback arises from the large number of bits required in the ROM, if one needs high precision of calculation. Furthermore the input combinational product terms of a ROM cannot be simplified. Lo et al. proposed the use of DGPLAs to avoid these shortcomings. They offered an algorithm to synthesize the DGPLAs by reducing the calculation of transformation, the location of the break points, and the ranges covered by the segments in an optimal condition. The same method can

be also applied to generating antilogarithms with not as good results though.

2.2 Logarithmic Arithmetic

2.2.1 Multiplication and Division

An algorithm for computer multiplication (division) using binary logarithms was described by Mitchell [Mit62]. A simple add (subtract) and shift operation was required. Hall et al. [Hal70] offered a refinement of this method, accompanied by an exhaustive error analysis of the product and quotient errors. A different approach was followed by Brubaker and Becker [Bru75]. They developed design curves for ROMs needed to generate the logarithm and antilogarithm transformations necessary for multiplication. The number of bits for a given accuracy was shown to be less than the one required for a direct multiply. A direct ROM-based multiplication requires one memory access while logarithmic multiplication requires two memory access times plus an addition. Similar algorithms along with examples and hardware implementations may be found in one or more of the following references: [Kin71, Li80, Lee77a, Lee77b, Lee79, Maj73, Swa75, Swa79, Tay84b, Tay85].

2.2.2 Addition and Subtraction

Kingsbury and Rayner [Kin71] pioneered in proposing two methods of adding or subtracting logarithmically encoded numbers. In the direct method of addition and subtraction, they use approximate evaluation of algorithms. However, their second method is based on table look-ups using read-only memory (ROM), which leads to potentially faster operations. The paper also indicates possible methods of table reduction. In 1975, Swartzlander and Alexopoulos [Swa75], unaware of previous work in the area proposed a sign/logarithm number system along with arithmetic algorithms identical to those proposed by Kingsbury and Rayner [Kin71]. Their paper suggests hardware architectures for arithmetic units and includes comparison of speeds with conventional arithmetic units. LNS was reinvented a third time in a paper by Lee and Edgar [Lee77a], but enriched with ideas and programs for implementation in 8 and 16-bit microcomputers. They examined more rigorously the LNS addition and subtraction algorithms with respect to storage, efficiency, accuracy, range, noise and array area [Lee79]. In a continued effort to research efficient algorithms in LNS, Li [Li80] presents four algorithms for addition and subtraction. This report compares table look-up and direct methods with interpolation and power series algorithms with respect to memory storage, speed and accuracy. Also covered in this

work is a comparative study of floating-point and logarithmic hardware.

2.2.3 Table Reduction Techniques

The most recent algorithms for implementation of addition or subtraction in a logarithmic environment call for table look-ups. The speed of operations then depends on the size of the table which is a function of the wordlength. A brute force table look-up is hindered by the monotonically decreasing nature of the logarithmic addition and the monotonically increasing nature of subtraction. A 8-bit implementation for example would require 2 K-bits of memory. Pioneering in this field too, Kingsbury and Rayner [Kin71] suggested two methods of reducing the storage requirements. One is to store in the table the values of the function corresponding to every 8th (say) value of the input and to interpolate linearly between these points. The other method is to store x at the address $f(x)$ instead of $f(x)$ at x , and then employ a read-and-compare successive approximation process. The current state of the art in designing high-speed ROMs or RAMs is such that LNS is useful for high-speed operations only when the wordlength is limited to 12 or 13 bits. The implementation of table look-up in hardware has been much delayed due to the prohibitive cost of memory. The realization of a custom-designed VLSI chip, though, brings into reality tables, which take advantage of the

properties of the function to be realized. A parallel-search table used for a pilot VLSI implementation of addition and subtraction algorithms is described by Bechtolsheim and Gross [Bec81]. The parallel-search table stores pairs of (x, y) values and searches the set of x -values for a matching interval. By searching digit-sequentially, instead of word-parallel, the look-up time of a parallel-search table is only proportional to the length of the x -values stored. Also since a read-only parallel-search table has single-transistor bit cells, it can be built as densely as a conventional read-only memory. As a result of discrete coding and the nature of the functions, a large portion of the output values is zero. Lee and Edgar [Lee79] determine a cutoff value, after which the input is mapped to a zero output value. The number of zeros depends on the number of fractional bits allotted in the word format and the chosen radix of the number system. Frey and Taylor [Fre85] suggest an interesting algorithm to reduce the size of the table, by recognizing the fact that the discrete encoding returns in a "staircase" type function, wherein a range of addresses will return the same output value. Taylor [Tay83a] proposes a linear interpolation scheme, which is practical from a hardware realization standpoint.

2.3 LNS Applications

2.3.1 Digital Filtering using LNS

The first (to my knowledge) application of LNS (other than the slide ruler) was a digital period-meter for a nuclear reactor designed by Eder et al. [Fur64]. Kingsbury and Rayner [Kin71] implemented a 2nd order recursive low-pass filter with 16-bit logarithmic arithmetic and demonstrated the improvement in dynamic range and performance over a 16-bit fixed-point filter. Kurokawa [Kur78] applied logarithmic arithmetic when implementing digital filters. Sicuranza offers some preliminary results on 2-D filter designs implemented with LNS, which confirm that it is possible to design, with sufficient accuracy, digital filters having their coefficients expressed as powers of some base a , and thus exploit the advantages offered by LNS [Sic81a-b, Sic82]. Finally [Sic83], he reports some design considerations based on the so-called "Implementation Cost," measured by the total number of bits required, and presents a few significant examples of 2-D filter approximation, including a 2-D wide-band differentiator, a circularly symmetrical half-plane low-pass filter and a 90° fan filter. In the same paper he offers a base optimization iterative procedure, proven to be influencing the convergence rate of the 2-D filters. Hall et al. [Hal70] apply LNS to a recursive digital filter, which fails to

compare favorably to an alternative design incorporating a cobweb array multiplier. However, such a favorable comparison is achieved in the case of a parallel digital filter bank with more than four subfilters. Finally they apply LNS to the multiplicative filters proposed by Oppenheim et al. [Opp68].

2.3.2 FFT Implementation

Swartzlander et al. [Swa83] presented a decimation in time sign/logarithm FFT. This is accompanied by an FFT error analysis and computer simulation results. It is shown that, besides being faster, LNS offers an improved error performance, when compared to conventional fixed or floating-point arithmetic.

2.3.3 Other LNS Applications

Swartzlander and Gilbert [Swa80a] analyzed the requirements for the new generation computed tomography machines and compared LNS convolution and weighted linear summation units to similar units implemented with merged arithmetic [Swa80b] and a two's complement modular array. They used a figure of merit relating processing speed to complexity and demonstrated that the LNS approach, which achieves extremely high dynamic range by use of constant relative precision, is the most efficient mechanization of the three examined. Kurokawa et al. [Kur80] applied LNS to recursive digital filters and performed an extensive error

analysis of the roundoff error accumulation, based on the assumption that the true (unrounded) result of the addition of two numbers is uniformly distributed between the lower and higher limits. Shenoy and Taylor [She82] designed a short term autocorrelation using LNS. In a later work [She83, She84] a theoretical error analysis of logarithmically implemented adaptive digital filters is presented. Simulation was used to compare them to their fixed-point counterparts with the same wordlength. LNS filters performed better. The most recent effort to design an LNS engine was made by Lang et al. [Lan85]. They presented integrated-circuit logarithmic arithmetic units, including adders, subtracters, multipliers and dividers. The design results were used to develop a size and speed comparison of integrated circuit logarithmic and fixed-point arithmetic units. Interestingly enough they chose an LNS base less than one, to restrict the number of product terms required by the PLA for the logarithmic adder or subtracter.

CHAPTER THREE

LNS AND LNS ARITHMETIC

The way arithmetic is performed using LNS is heavily dependent on the method chosen to represent the real numbers and the base of the logarithms. Several number representations have been proposed. In this dissertation the most widely adopted representation has been chosen and a proof that the choice of base is immaterial is given. Several basic LNS arithmetic operations have been also outlined.

3.1 LNS Description

Many different schemes have been adopted to represent numbers in LNS. In several works [Swa75, Lee77b] , a sign/logarithm representation has been chosen, where the LNS exponent is a fixed-point number with an integer and a fractional part. Both the number and the exponent are signed. The number is represented in sign magnitude and the exponent in offset binary number system. This way any negative logarithms, causing problems in further computations, are avoided. This introduces some overhead when multiplication and division are performed and therefore it is not used here.

In most of the work regarding theory and implementation of LNS, the base of the system is selected to be $r = 2$. This offers obvious advantages, but some researchers [Lan85] have proposed bases that are between 0 and 1, guaranteeing that the LNS exponent will be positive, when its magnitude is absolutely bounded by 1, as is the case with the signals. In general, this requires prenormalization of the data. Some others [Sic83] have offered base optimization iterative procedures, which are positively affecting the applications they are considered for. An error criterion has been established in section 3.1.2 of this dissertation, according to which a base optimization answer is derived.

3.1.1 Logarithmic Number System Representation

For a real number $X = \pm r^{\pm x}$, its LNS representation can be given by a $(N+2)$ -bit exponent word of the form



where S_X is the overall sign of the real number X and S_x is the sign of the exponent. This way it is possible to represent logarithmically even negative numbers. Computation of S_X is operation dependent, requiring support from comparators and/or multiplexers and/or XOR

gates. Computation of S_x can be performed as part of the computation of the LNS exponent itself. Of course, if internal exponent representation (sign/magnitude for example) does not allow for automatic sign computation, again comparators and gates have to be provided for that. From the N weighted bits of the exponent, F bits are assigned as fractional bits and $N - F = I$ bits are assigned to be the integer bits. More specifically, the absolute value of the LNS exponent x is given by

$$x = \sum_{i=0}^{N-1} a_i 2^{(i-F)} ; \quad a_i \in \mathbb{Z}_2 \quad (3.1)$$

By distributing the appropriate number of bits for the integer and fractional parts, both large dynamic range and high precision can be achieved. More specifically, this system is characterized by

- Largest positive magnitude : $|X|_{\max} = r^{(2^{-F}(2^N-1))} \approx r^{2^I}$
- Smallest positive magnitude: $|X|_{\min} = r^{(-2^{-F}(2^N-1))} \approx r^{-2^I}$
- Range = Largest/Smallest : $r^{(2^{-F+1}(2^N-1))} \approx r^{2^{I+1}}$

So for example, if $I = 6$, $F = 9$, and $r = 2$, one finds

$$5.42 \times 10^{-20} \leq |X| \leq 1.84 \times 10^{19} ; \quad \text{Range} \approx 3.39 \times 10^{38}$$

Observe that, like in floating-point systems, there is a "dead zone" around zero, which can be closed by increasing the number of fractional bits of the LNS exponent.

3.1.2 LNS Base Optimization

Theorem

If a) the dynamic range to be covered by the logarithmic number system is $\pm V$, b) the LNS wordlength (in bits) is N and c) for a random variable x representing the LNS exponents, a model for the maximum error budget is described by the function $E(r) = r^x (r^{e(r)} - 1)$ with $e(r) = 2^{-F(r)}$, where $F(r)$ is the number of fractional bits required for the specific case of base selection, then there is no real base minimizing the error function.

Proof

Suppose that for a certain base r , the number of integer bits available is I' , whereas it is I for base = 2. Then the dynamic range is expressed as $V = 2^{2^I} = r^{2^{I'}}$. Then

$$2^I = 2^{I'} \lg r \rightarrow I = I' + \lg(\lg r) \rightarrow$$

$$I' = I - \lg(\lg r) \rightarrow F(r) = N - I' = N - I + \lg(\lg r)$$

To minimize $E(r)$, the derivative is evaluated

$$\frac{dE(r)}{dr} = \frac{d[r^x (r^{e(r)} - 1)]}{dr} = \frac{d[r^x (r^{2^{-F(r)}} - 1)]}{dr}$$

$$\begin{aligned}
&= \frac{d(r^x)}{dr} \left[r^{2^{-F(r)}} - 1 \right] + r^x \frac{d \left[r^{2^{-F(r)}} - 1 \right]}{dr} \\
&= x r^{x-1} \left[r \left(2^{-N+I - \lg(\lg r)} \right) - 1 \right] + r^x \frac{d \left[r^{2^{-F(r)}} \right]}{dr} \\
&\quad \longleftarrow G(r) \longrightarrow \\
&= G(r) + r^x \left[2^{-F(r)} r \left(2^{-F(r)} - 1 \right) + \log_2 r \cdot 2^{-F(r)} \cdot 2^{-F(r)} \log_2 \frac{d(-F(r))}{dr} \right] \\
&= G(r) + r^x 2^{-F(r)} r^{2^{-F(r)}} \left[\frac{1}{r} - \log_2 \log_2 \left(\frac{1}{\log_2 \log_2 \frac{\log r}{\log 2} r} \right) \right] \\
&= G(r) + r^x 2^{-F(r)} r^{2^{-F(r)}} \frac{1}{r} [1 - 1] \\
&= G(r) \quad \text{for all } r.
\end{aligned}$$

For $G(r)$ to be zero, then $2^{-N+I - \lg(\lg r)} = 0 \quad \rightarrow$

$$-N + I - \lg(\lg r) = -\infty \quad \rightarrow \quad r = \infty.$$

In other words, there is no real base r minimizing the considered criterion of optimality. However, the factor $(r^{e(r)} - 1)$ by which every real number $x = r^x$ is multiplied assumes the same value for all real bases r . Any base, like $r = 2$, $r = e$, $r = 10$, etc. can be used without altering the error budget. Consequently, the field is open for application of any other optimization schemes, imposed by hardware or software requirements. The above analysis was verified experimentally with some of the test values reported in Table 3.1, where the various

parameters have the following values:

$$N = 12, \quad V = 2^{16} = 2^{2^4} \rightarrow$$

$$I = 4 \quad \text{and} \quad F(r) = 12 - 4 + \lg(\lg r)$$

TABLE 3.1
Error Values for Several LNS Bases and
for the Same Dynamic Range

r	$F(r)$	$r^{e(r)} - 1$
16	10.0	2.711275×10^{-3}
4	9.0	2.711275×10^{-3}
2	8.0	2.711275×10^{-3}
$\sqrt{2}$	7.0	2.711275×10^{-3}
e	8.5287	2.711275×10^{-3}

3.2 LNS Arithmetic Operations

Following the previously adopted number representation, arithmetic in LNS is described below. For the fastest execution of operations like addition or subtraction, table look-up operations have been employed to support them. In the following discussion of operations the numbers are represented as

$$A \rightarrow S_A r^a, \quad B \rightarrow S_B r^b, \quad C \rightarrow S_C r^c, \quad I \rightarrow S_I r^s, \quad \Delta \rightarrow S_\Delta r^\delta$$

$$H \rightarrow S_H r^h, \quad E \rightarrow S_E r^e, \quad M \rightarrow S_M r^\mu, \quad T \rightarrow S_T r^\tau, \quad V \rightarrow S_V r^v$$

3.2.1 Multiplication and Division

$$\text{For } C = A \times B \quad \rightarrow \quad c \leftarrow a + b \quad \text{and } S_C = S_A \oplus S_B \quad (3.2)$$

$$\text{For } C = A \div B \quad \rightarrow \quad c \leftarrow a - b \quad \text{and } S_C = S_A \oplus S_B$$

where \oplus denotes an "exclusive or" operation. It is obvious that multiplication and division in the conventional systems are substituted by addition and subtraction in LNS, resulting in very fast operations. Overflows and underflows can be very easily detected, by simple comparisons with the largest and the smallest numbers representable by the system. The flowchart of the LNS multiplication/division is shown in Figure 3.1.

3.2.2 Addition and Subtraction

Without loss of generality one can assume that $A \geq B$.

Then

$$\begin{aligned} \text{for } \Sigma = A + B \quad \rightarrow \quad s \leftarrow a + \Phi(b - a) \quad \text{and } S_\Sigma = S_A \\ \text{with } \Phi(b - a) = \text{lr}(1 + r^{b-a}) \\ \text{For } \Delta = A - B \quad \rightarrow \quad \delta \leftarrow a + \Psi(b - a) \quad \text{and } S_\Delta = S_A \\ \text{with } \Psi(b - a) = \text{lr}(1 - r^{b-a}) \end{aligned} \quad (3.3)$$

For the special case when $A = B$, then $\delta \leftarrow 0$, and $S_\Delta = 0$.

The flowchart for LNS addition/subtraction is offered in Figure 3.2.

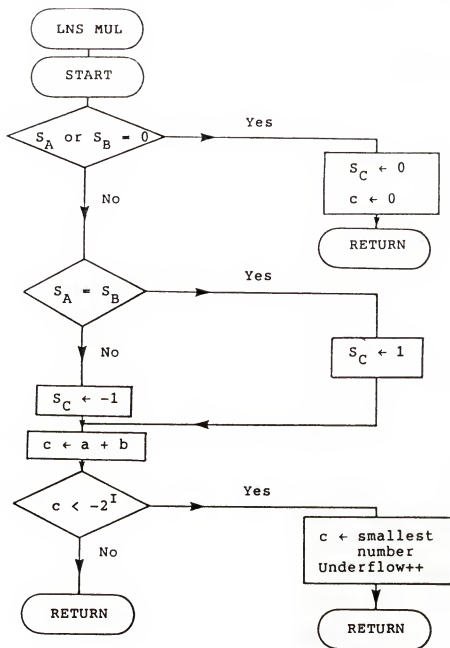


FIGURE 3.1
Flowchart for LNS Multiplication/Division

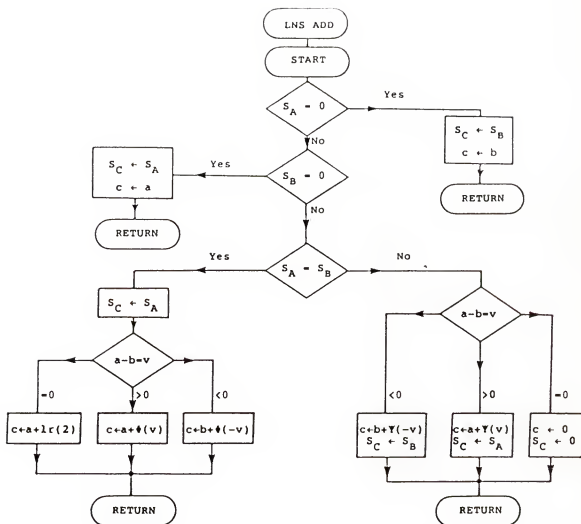


FIGURE 3.2
Flowchart for LNS Addition/Subtraction

Historically, addition and subtraction have been the principal obstacles to developing versatile LNS engines. It can be seen that both operations involve logarithmic mappings and regular additions as well. The values of $\Phi(v)$ and $\Psi(v)$, where $v = |a - b|$, can be obtained as memory table look-ups. Based on contemporary high-speed memory chips limitations (see Table 3.2) the wordlength of v can be estimated to be on the order of 12 bits. This has been the historical limit to its precision. Nevertheless, it has been shown to be potentially powerful in the class of digital filtering problems, in which the speed, nature of signals and component count offset any accuracy considerations. For example, real time digital filtering of radar video for moving target detection, synthetic aperture processing, and pulse compression are in this class. Because of the statistical nature of the sampled signals, the large amount of signal integration required and the characterization of detection performance on a probabilistic basis, the accuracy of a single computation has less importance than the mean and variance of the operation on the signal ensemble. Radar video is characterized by broad bandwidth and corresponding high data rates, which make real time multiplication with readily available logic very difficult. Furthermore, multiple filters are usually required because the noise is colored and the filter bandpass is but a small fraction of the actual signal bandwidth. Examples like the one just

TABLE 3.2
Commercially Available High-Speed Memory Chips

	Size	Access Time (ns)	Technology Family	Pins
P R O M S	1K × 4	35	TTL	18
	2K × 4	35	TTL	18
	1K × 8	30	TTL	24
	2K × 8	35	TTL	24
	4K × 4	35	TTL	20
	4K × 8	40	TTL	24
	8K × 8	40	TTL	24
S R A M S	1K × 4	8	ECL	24
		30	TTL	16
	4K × 1	15	ECL	18
		10	ECL	20
		55	CMOS	18
	2K × 8	45	CMOS	20
	4K × 4	15	ECL	28
		55	CMOS	20
	8K × 8	55	CMOS	20
	16K × 1	15	ECL	20
		35	CMOS	20
	16K × 4	35	ECL	20
	64K × 1	55	CMOS	22
		40	NMOS	22
	64K × 8	55	TTL	22

Data are taken from the DATABOOKS 84/85 of:
AMD, FAIRCHILD, NEC, HITACHI, MOTOROLA

given really prove that 'even precision "handicapped" LNS engines are valuable in certain cases.

PLAs have been also used in place of ROMs and RAMs by some designers. An 8-bit single chip 3 μ m CMOS that makes extensive use of PLAs has been reported [Lan85].

3.2.3 Other Arithmetic Operations

Some more operations include

Squaring : For $T = A^2 \rightarrow \tau \leftarrow 2a$ and $S_T = S_A$

Square rooting : For $C = A^{\frac{1}{2}} \rightarrow c \leftarrow \frac{a}{2}$ and $S_C = S_A$

Exponentiating : For $E = A^B \rightarrow \varepsilon \leftarrow ar^b$ and

$$S_E = \begin{cases} S_A & \text{for } A \geq 0 \text{ or } (A < 0 \text{ and } B \text{ odd}) \\ -S_A & \text{for } A < 0 \text{ and } B \text{ even.} \end{cases}$$

(3.4)

The exponents of some hyperbolic trigonometric functions are given as

Hyp. cosine : For $H = \cosh X \rightarrow h \leftarrow z + \phi(2z) - 1r2$

Hyp. sine : For $M = \sinh X \rightarrow \mu \leftarrow z + \Psi(2z) - 1r2$

Hyp. tangent : For $T = \tanh X \rightarrow \tau \leftarrow \Psi(2z) - \phi(2z)$

Hyp. cotangent: For $C = \coth X \rightarrow c \leftarrow \phi(2z) - \Psi(2z)$ (3.5)

Hyp. secant : For $A = \operatorname{sech} X \rightarrow a \leftarrow -z - \phi(2z) + 1r2$

Hyp. cosecant : For $V = \text{csch } X \rightarrow v \leftarrow -z - \Psi(2z) + 1r2$

$$\text{with } z = \frac{X}{10gr} = X \text{ } 1r(e) = 1r(e^X).$$

For a derivation of the above relationships see Appendix A. They bring into light some of the advantages of LNS. For example, squaring and square rooting require but a mere shift of the LNS exponent to the left or to the right respectively. A basic LNS architecture, based on the above equations, is shown in Figure 3.3. The computation of regular and hyperbolic trigonometric functions proves to be extremely simple and without imposing extra hardware burdens. Required is only an extra table for the generation of z (defined in equations (3.5)). After z is generated, it will be shifted by 1 bit and presented to the tables Φ and Ψ , according to the function that has to be implemented. For a stand-alone or integrated hyperbolic trigonometric LNS processor the complex trigonometric functions would not require more time than a LNS addition. In Figure 3.4 the architecture of an LNS hyperbolic trigonometric processor is given. It can be seen that this design can be easily integrated with the basic LNS architecture offered in Figure 3.3. A simple CORDIC replacement LNS alternative architecture is also described in Chapter Nine (see Figure 9.2). These two architectures combined form the basis for a complete LNS trigonometric processor. The regularity of the data flow is remarkable and well suited for the VLSI design of LNS trigonometric processor.

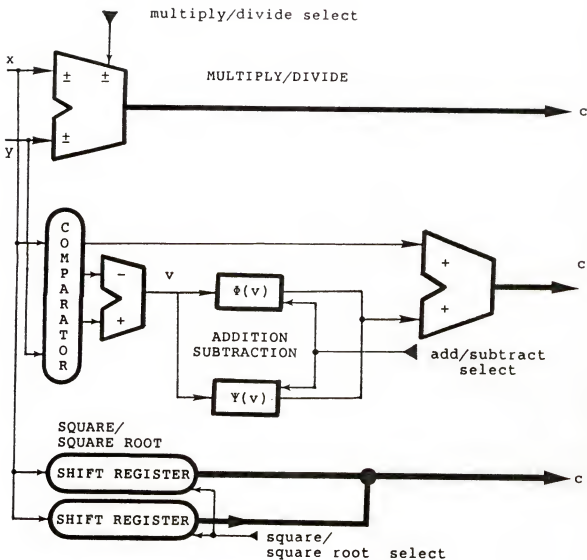


FIGURE 3.3
Basic LNS Arithmetic Unit

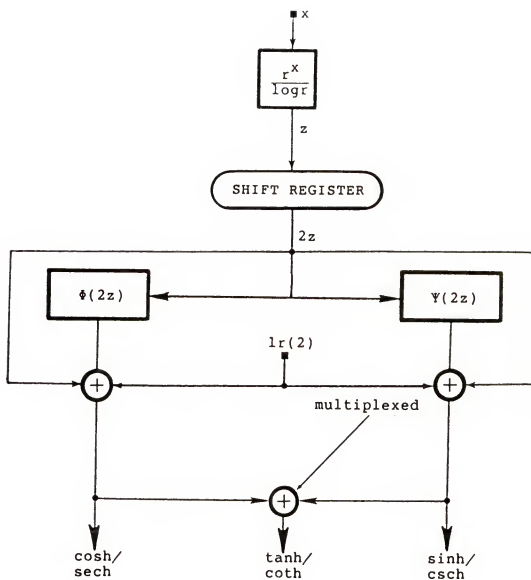


FIGURE 3.4
LNS Hyperbolic Trigonometric Processor

CHAPTER FOUR CONVERSION TO AND FROM THE LNS

In previous sections, a case was made to justify the LNS processor as a special-purpose processor. In some cases, it can be used directly after the acquisition of the signal, without any number system conversion. Another case is where the LNS processor is used as a specialized arithmetic machine. Such is the case of a stand-alone divide unit, or a trigonometric processor (CORDIC replacement unit). This will be discussed in later chapters. In a dedicated LNS architecture, data need only be converted to and from LNS at the input/output boundary. Within the LNS machine, data are manipulated in a LNS format and do not require that a big conversion overhead penalty be paid. A typical example of such a case is the hybrid floating-point and logarithmic processor (FU)² [Tay85], which is shown in Figure 4.1. Floating-point number representation is the dominating choice of systems designers, when a large dynamic range and high precision are simultaneously required. However, there are two problems associated with this kind of arithmetic. Besides being slow, it offers a hardware utilization of as low as 50 percent. This is the result of two different addition and multiplication paths. The (FU)² consists of three

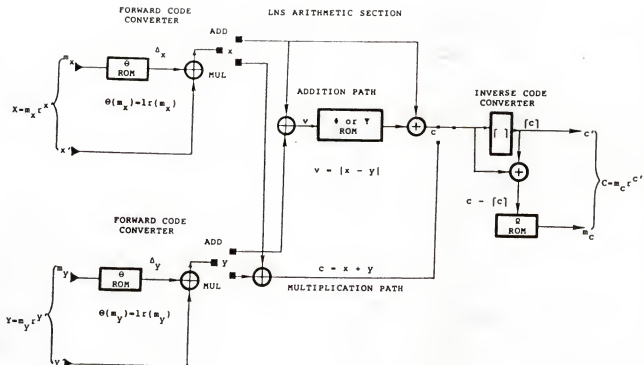


FIGURE 4.1
Hybrid Floating-Point Logarithmic Processor

architecturally distinct sections: a) the FLP to LNS converter, b) the LNS arithmetic unit, and c) the LNS to FLP converter. It has been shown to possess a number of attractive attributes:

- No alignment is required for addition or subtraction.
- The multiplier is a subset of the adder.
- The data flow is highly regular, because all operations occupy the same interval of time, regardless of the relative values of the operands.

4.1 Conversion from FLP to LNS

In a floating-point environment, a real number X can be approximated as

$$X = S_X m_X r^{x'} ; \quad \frac{1}{r} < m_X < 1 \quad (4.1)$$

where S_X is the sign of the number, r is the base, m_X is the unsigned M -bit mantissa and x' is the $(E+1)$ -bit signed exponent. For a $X = S_X r^x$ LNS representation, discussed in Chapter Three, with an $(N+2)$ -bit exponent and a memory table Θ offering at its output the logarithm to the base r of its input, x is given by

$$x = x' + \Delta x ; \quad \Delta x = \Theta(m_X) \quad (4.2)$$

If $r = 2$, then $0.5 \leq m_X \leq 1$ and $\Delta x \leq 0$.

Several conversion schemes have been proposed and implemented. The most recent one, offered by Lo and Aoki

[Lo85], uses table look-ups. It employs Difference Grouping Programmable Logic Array (DGPLAs) to simplify the number of input combinational product terms. The architecture of the FLP to LNS converter is shown in Figure 4.2. For a base $r = 2$, the presence of the adder is not really necessary, since Δx will always range between -1 and 0 . Therefore, the memory table can be programmed to directly output the fractional part of the the 2's complement version of Δx , which can then be concatenated to $x'-1$ in order to result in x . Of course, to form $x'-1$ requires one add time but this operation may be postponed to the next stage of the (FU)². There, if the operation to be performed is division, the operation $x-1$ can be totally eliminated, since division of real numbers requires subtraction of the two LNS exponents involved. Therefore, for a divide-only unit, the conversion time from FLP to LNS is only one look-up, instead of one look-up plus one add that is required in general. An error analysis (based on the assumption that the digital number presented to the DGPLA is the original one) is also offered in Lo and Aoki [Lo85]. An error analysis, not restricted by the above assumption is presented in the next section. It results in an analytical expression for the probability density function (p.d.f.) of the conversion error.

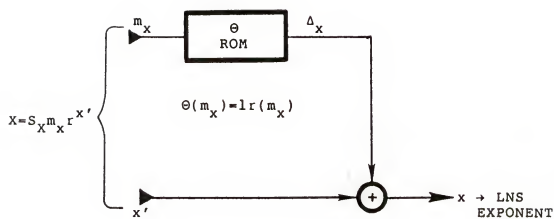


FIGURE 4.2
Architecture of an FLP to LNS Converter

4.2 Error Analysis

The only source of error in forming x is the one resulting from the logarithmic mapping $\Delta x = \Theta(m_x)$, which is performed as a memory table look-up operation. The proposed error analysis model is illustrated in Figure 4.3. There, the two paths associated with the forming of Δx and its finite wordlength approximation are shown as parallel ones. Upon receiving the FLP mantissa m_x , the lower path provides the ideal mapping $\Delta x = \Theta(m_x)$, while the upper part consists of an Input Quantizer (QI, which provides a discrete value of m_x denoted m_x^*); then an ideal mapping of m_x^* into $L = \Theta(m_x^*)$ and finally an Output Quantizer (QO, which provides the machine version of L , namely $\Delta x^* = \text{RND}[L]$). The input and output quantization errors are then defined as

$$E_I = m_x - m_x^* ; \quad E_O = \Theta(m_x^*) - [\text{RND}[\Theta(m_x^*)]] \quad (4.3)$$

where E_I and E_O are uniform white noise sequences possessing the following statistical properties:

$$E[m_x E_I] = E[\Theta(m_x^*) E_O] = 0 ; \quad E[E_{Ik} E_{Ij}] = \frac{q_I^2}{12} \delta_{kj} \quad (4.4)$$

$$E[E_I] = E[E_O] = 0 ; \quad E[E_{Ok} E_{Oj}] = \frac{q_O^2}{12} \delta_{kj}$$

Here, $q_I = 2^{-N_I}$ and $q_O = 2^{-N_O}$, where N_I and N_O are the numbers of bits available at the input and output of the

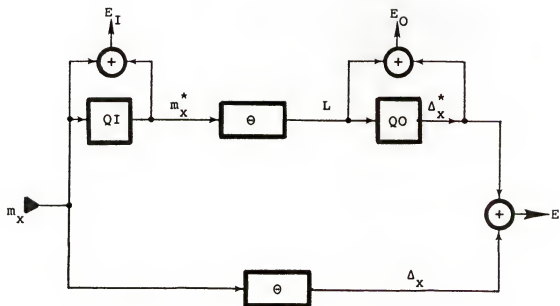


FIGURE 4.3
Error Model for FLP to LNS Conversion

logarithmic mapping L , and δ_{kj} is the Kroenecker delta. In addition we assume that $m_x \in U\left[\frac{1}{r}, 1\right)$. The final error metric E is found by using the parameter $D = \frac{E_I}{m_x}$ as

$$E = \Delta x^* - \Delta x = \Theta(m_x + E_I) + E_O - \Theta(m_x) = \ln(1+D) + E_O \quad (4.5)$$

Through the application of theory of a function of a random variable [Pap65], the p.d.f. of E is found to be

$$\begin{aligned} f_D(D) &= \int_{-\infty}^{\infty} |m_x| f_{E_I, m_x}(Dm_x, m_x) dm_x \\ &= \int_{-\infty}^{\infty} |m_x| f_{E_I}(Dm_x) f_{m_x} dm_x \\ &= 2 \int_{\frac{1}{2}}^1 m_x f_{E_I}(Dm_x) dm_x \end{aligned} \quad (4.6)$$

or

$$f_D(D) = \begin{cases} \frac{2}{q_I} \int_{\frac{1}{2}}^{-\frac{q_I}{2D}} m_x dm_x & \text{for } -q_I \leq D \leq -\frac{q_I}{2} \\ \frac{2}{q_I} \int_{\frac{1}{2}}^1 m_x dm_x & \text{for } -\frac{q_I}{2} \leq D \leq 0 \\ \frac{2}{q_I} \int_{\frac{1}{2}}^1 m_x dm_x & \text{for } 0 \leq D \leq \frac{q_I}{2} \\ \frac{2}{q_I} \int_{\frac{1}{2}}^{\frac{q_I}{2D}} m_x dm_x & \text{for } \frac{q_I}{2} \leq D \leq q_I \end{cases} \quad (4.7)$$

or

$$f_D(D) = \begin{cases} \frac{q_I}{4D^2} - \frac{1}{4q_I} & \text{for } -q_I \leq D \leq -\frac{q_I}{2} \\ \frac{3}{4q_I} & \text{for } -\frac{q_I}{2} \leq D \leq \frac{q_I}{2} \\ \frac{q_I}{4D^2} - \frac{1}{4q_I} & \text{for } \frac{q_I}{2} \leq D \leq q_I \end{cases} \quad (4.8)$$

For a better understanding of the algebra involved, the following parameters are defined

$$J = \text{lr} \left(1 - q_I \right) \quad K = \text{lr} \left(1 - \frac{q_I}{2} \right) \quad (4.9)$$

$$M = \text{lr} \left(1 + \frac{q_I}{2} \right) \quad N = \text{lr} \left(1 + q_I \right)$$

$$\text{and } T^\pm = T \pm \frac{q_I}{2} \quad \text{for } T = J, K, M, N \quad \text{and } E^\pm = E \pm \frac{q_O}{2}.$$

Proceeding, a new random variable P is defined as

$$P = \text{lr}(1 + D).$$

Then $D = r^P - 1$ and

$$f_P(P) = \frac{f_D(D)}{\left| \frac{\partial P}{\partial D} \right|_D} = r^P \log r f_D(D) = r^P \log r f_D(r^P - 1) \quad (4.10)$$

or

$$f_P(P) = \begin{cases} r^P \log r \left[\frac{q_I}{4(r^P - 1)^2} - \frac{1}{4q_I} \right] & \text{for } J \leq P \leq K \\ r^P \log r \frac{3}{4q_I} & \text{for } K \leq P \leq M \\ r^P \log r \left[\frac{q_I}{4(r^P - 1)^2} - \frac{1}{4q_I} \right] & \text{for } M \leq P \leq N \end{cases} \quad (4.11)$$

Assuming that P and E_0 are statistically independent random variables, one can find for the final error $E = P + E_0$ that

$$f_E(E) = \int_{-\infty}^{\infty} f_P(P) f_{E_0}(E-P) dP \quad (4.12)$$

or

$$\begin{aligned} f_E(E) = & \int_J^K A(P) f_{E_0}(E-P) dP + \\ & \int_K^M B(P) f_{E_0}(E-P) dP + \\ & \int_M^N A(P) f_{E_0}(E-P) dP \end{aligned} \quad (4.13)$$

$$\text{with } A(P) = r^P \log r \left[\frac{q_I}{4(r^P - 1)^2} - \frac{1}{4q_I} \right]; \quad B(P) = \frac{3}{4q_I} r^P \log r \quad (4.14)$$

$$\int_a^b A(P) dP = \frac{q_I}{4} \left[\frac{r^a - r^b}{(1-r^b)(1-r^a)} + \frac{r^a - r^b}{4q_I} \right]; \quad \int_a^b B(P) dP = \frac{3(r^b - r^a)}{4q_I}$$

where a, b are defining the domains for $A(P)$ and $B(P)$. Depending upon the relative positions of a, b and the limits of the integrals in Equation (4.13), one can distinguish between the following cases:

Case i) $q_0 < q_I$.

Then by applying some straightforward algebra one finds

$$f_E(E) = \left\{ \begin{array}{ll} \frac{1}{q_0} \int_J^{E^+} A(P) dP & \text{for } J^- \leq E \leq J^+ \\ \frac{1}{q_0} \int_{E^-}^{E^+} A(P) dP & \text{for } J^+ \leq E \leq K^- \\ \frac{1}{q_0} \int_{E^-}^K A(P) dP + \int_K^{E^+} B(P) dP & \text{for } K^- \leq E \leq K^+ \\ \frac{1}{q_0} \int_{E^-}^{E^+} B(P) dP & \text{for } K^+ \leq E \leq M^- \quad (4.15) \\ \frac{1}{q_0} \int_M^{E^+} A(P) dP + \int_{E^-}^M B(P) dP & \text{for } M^- \leq E \leq M^+ \\ \frac{1}{q_0} \int_{E^-}^{E^+} A(P) dP & \text{for } M^+ \leq E \leq N^- \\ \frac{1}{q_0} \int_{E^-}^N A(P) dP & \text{for } N^- \leq E \leq N^+ \end{array} \right.$$

or after substituting for the values of the integrals involved from equation (4.14)

$$f_E(E) = \begin{cases} \frac{q_I}{4q_O} \left[\frac{1}{1-r^{E^+}} - \frac{1}{q_I} \right] - \frac{1}{4q_I q_O} \left[r^{E^+} + q_I - 1 \right] ; & J^- \leq E \leq J^+ \\ \frac{q_I}{4q_O} \left[\frac{1}{1-r^{E^+}} - \frac{1}{1-r^{E^-}} \right] - \frac{1}{4q_I q_O} \left[r^{E^+} - r^{E^-} \right] ; & J^+ \leq E \leq K^- \\ \frac{q_I}{4q_O} \left[\frac{2}{q_I} - \frac{1}{1-r^{E^-}} \right] - \frac{1}{4q_I q_O} \left[1 - \frac{q_I}{2} - r^{E^-} \right] + \\ \quad + \frac{3}{4q_I q_O} \left[r^{E^+} - 1 + \frac{q_I}{2} \right] ; & K^- \leq E \leq K^+ \end{cases} \quad (4.16a)$$

$$f_E(E) = \begin{cases} \frac{3}{4q_I q_O} \left[r^{E^+} - r^{E^-} \right] ; & K^+ \leq E \leq M^- \end{cases} \quad (4.16b)$$

$$f_E(E) = \begin{cases} \frac{q_I}{4q_O} \left[\frac{2}{q_I} + \frac{1}{1-r^{E^+}} \right] - \frac{1}{4q_I q_O} \left[r^{E^+} - 1 - \frac{q_I}{2} \right] + \\ \quad + \frac{3}{4q_I q_O} \left[1 + \frac{q_I}{2} - r^{E^+} \right] ; & M^- \leq E \leq M^+ \\ \frac{q_I}{4q_O} \left[\frac{1}{1-r^{E^+}} - \frac{1}{1-r^{E^-}} \right] - \frac{1}{4q_I q_O} \left[r^{E^+} - r^{E^-} \right] ; & M^+ \leq E \leq N^- \\ \frac{q_I}{4q_O} \left[-\frac{1}{1-r^{E^-}} - \frac{1}{q_I} \right] - \frac{1}{4q_I q_O} \left[1 - r^{E^+} + q_I \right] ; & N^- \leq E \leq N^+ \end{cases} \quad (4.16c)$$

This complicated piecewise linear continuous expression can be approximated by using two distinct first-order polynomial approximations. The simplified p.d.f. is then

given by the curve

$$f_E(E) = \begin{cases} \lambda_1 \frac{1}{K^+ - J^-} [E - J^-] & \text{for } J^- \leq E \leq K^+ \\ \frac{3}{4q_I q_O} \left[r^{E^+} - r^{E^-} \right] & \text{for } K^+ \leq E \leq M^- \\ \lambda_2 \frac{1}{M^- - N^+} [E - N^+] & \text{for } M^- \leq E \leq N^+ \end{cases} \quad (4.17)$$

$$\text{with } \lambda_1 = \frac{3 \left[1 - \frac{q_I}{2} \right] \left[r^{q_O - 1} \right]}{4q_I q_O} ; \quad \text{and} \quad \lambda_2 = \frac{3r^{\left(\frac{q_O}{2} \right)} \left[1 + \frac{q_I}{2} \right] \left[r^{q_O - 1} \right]}{4q_I q_O}$$

The precise p.d.f. for the final error is depicted by the solid line in Figure 4.4. The approximating curve (generated by the code found in Appendix F) is outlined in Figure 4.4 by the dotted line. Simulation was also used to verify the theoretical results. The histogram of the error E was obtained first. Next the histogram values were divided by the total number of samples and the step of the histogram. The result was compared to that of the precise theoretical curve obtained for the p.d.f. of the error. The two curves remarkably coincided. The interrupted line in Figure 4.4 stands for the experimental curve. The simulation was run on a VAX 11-750 machine and the total number of samples was taken to be 30,000 for all experiments. The p.d.f. curves for $N_I = 7$ and $N_O = 8$ are shown in Figure 4.4. The program used for the simulation of the logarithmic encoder is listed in Appendix B, while

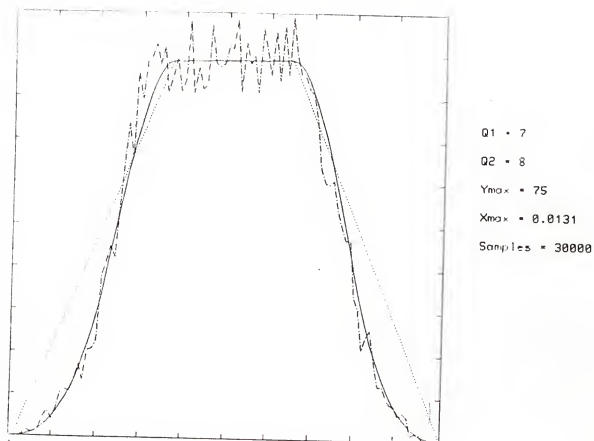


FIGURE 4.4
P.d.f. for the Error in FLP to LNS Conversion. Case i)

the one used for the graphical depiction of the theoretical curve and comparisons with the experimental one is listed in Appendix C.

Case ii) $q_0 = q_I$.

Then, omitting the intermediate calculations one finds for $f_E(E)$

$$f_E(E) = \begin{cases} \frac{1}{q_0} \int_J^{E^+} A(P) dP & \text{for } J^- \leq E \leq K^- \\ \frac{1}{q_0} \int_J^K A(P) dP + \int_K^{E^+} B(P) dP & \text{for } K^- \leq E \leq J^+ \\ \frac{1}{q_0} \int_{E^-}^K A(P) dP + \int_K^{E^+} B(P) dP & \text{for } J^+ \leq E \leq K^+ \\ \frac{1}{q_0} \int_{E^-}^{E^+} B(P) dP & \text{for } K^+ \leq E \leq M^- \quad (4.18) \\ \frac{1}{q_0} \int_M^{E^+} A(P) dP + \int_{E^-}^M B(P) dP & \text{for } M^- \leq E \leq N^- \\ \frac{1}{q_0} \int_M^N A(P) dP + \int_{E^-}^M B(P) dP & \text{for } N^- \leq E \leq M^+ \\ \frac{1}{q_0} \int_{E^-}^N A(P) dP & \text{for } M^+ \leq E \leq N^+ \end{cases}$$

Again, the above theoretical curve (generated by the code found in Appendix D and represented by the solid line in Figure 4.5) was tested against the experimental one, obtained for a sample size of 20,000 for $N_I = N_O = 12$, and

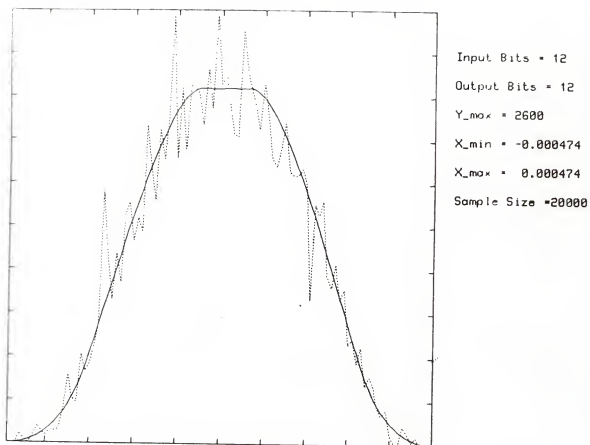


FIGURE 4.5
P.d.f. for the Error in FLP to LNS Conversion. Case ii)

the result is absolute coincidence.

Case iii) $q_0 = 1 + q_I$.

Again, omitting the intermediate calculations one finds for $f_E(E)$

$$f_E(E) = \left\{ \begin{array}{ll} \frac{1}{q_0} \int_J^{E^+} A(P) dP & \text{for } J^- \leq E \leq K^- \\ \frac{1}{q_0} \int_J^K A(P) dP + \int_K^{E^+} B(P) dP & \text{for } K^- \leq E \leq J^+ \\ \frac{1}{q_0} \int_{E^-}^K A(P) dP + \int_K^{E^+} B(P) dP & \text{for } J^+ \leq E \leq M^- \\ \frac{1}{q_0} \int_{E^-}^K A(P) dP + \int_K^{E^+} B(P) dP + \int_M^{E^+} A(P) dP & \text{for } M^- \leq E \leq K^+ \\ \frac{1}{q_0} \int_M^{E^+} A(P) dP + \int_{E^-}^M B(P) dP & \text{for } K^+ \leq E \leq N^- \\ \frac{1}{q_0} \int_M^N A(P) dP + \int_{E^-}^M B(P) dP & \text{for } N^- \leq E \leq M^+ \\ \frac{1}{q_0} \int_{E^-}^N A(P) dP & \text{for } M^+ \leq E \leq N^+ \end{array} \right. \quad (4.19)$$

Again the above theoretical curve (generated by the code found in Appendix E and represented by the solid line in Figure 4.6) was tested against the experimental one,

obtained for a sample size of 30,000 for $N_I = 8$, and $N_O = 7$, and the result is absolute coincidence. This last case is of particular interest, since the precision offered by QI is higher than that of QO ($N_I > N_O$) by at least 1 bit in the case of $r = 2$. This is a result of the fact that the leading bit of the mantissa is always 1, and therefore can be omitted from being presented to the table.

4.3 Conversion from LNS to FLP

Again, software routines, as perhaps the ones found in [Che72], can be used to convert an LNS exponent to a FLP number. A hardware realization would also require a table look-up operation to assist in mapping x to $X = m_x r^{x'}$. More specifically

$$x' \leftarrow [x] ; \quad m_x \leftarrow Q(x' - x) \quad \text{where} \quad Q(t) = r^t$$

The LNS to FLP conversion architecture is shown in Figure 4.7. Of course, $[x] - x$ is $1 - x_F$, where x_F is the fractional part of x . The computation of $1 - x_F$ is equivalent to computing the 2's complement of x_F or $1 + \overline{x_F}$. As in the case of FLP to LNS conversion, for $r=2$ the addition is not necessary, if the memory is programmed to output directly $Q(x+1)$, rather than $Q(x)$. The ceiling function, which is necessary to form $[x]$, based on an incrementer is fast enough not to be critical to the timing. The conversion procedure, then, takes one table look-up time. Software interpolation is not recommended

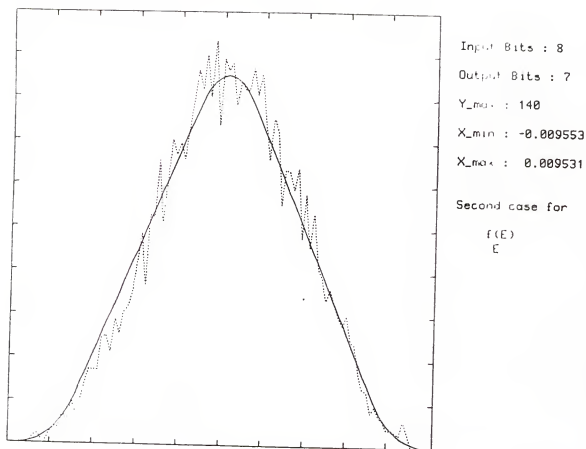


FIGURE 4.6
 P.d.f. for the Error in FLP to LNS Conversion. Case iii)

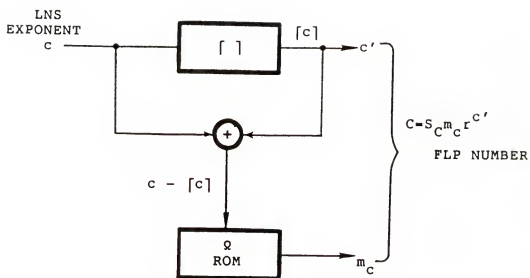


FIGURE 4.7
Architecture of an LNS to FLP Converter

in this case, since the implementation of the exponential function is slower than that of the log function. Using Chen's algorithm for example, it would require 12 add and 6 look-up times versus only 1 look-up time (of a larger and consequently slower table though) when using a ROM.

CHAPTER FIVE

MEMORY TABLE REDUCTION TECHNIQUES

5.1 Essential Zeros

The LNS described in Chapter Three is shown to be memory table look-up intensive, when it comes to performing additions or subtractions. There, the memory table address is defined as the absolute difference of the LNS exponents of the two operands. For x , y being the exponents to be logarithmically added and $v = |x - y|$, the tables must yield the values of

$$\Phi(v) = \log_r(1 + r^{-v}) \text{ and } \Psi(v) = \log_r(1 - r^{-v})$$

By demanding that v is always positive, the values of Φ and Ψ are always less than one and therefore they require an F -bit representation instead of a N -bit one ($N = I + F$). In Figure 5.1 the curve $\Phi(v)$ versus v is shown, whereas Figure 5.2 shows the curve $\Psi(v)$ versus v for various values of the base r . It can be observed that a smaller value of r results in a larger table.

The (absolutely) monotonically decreasing nature of the two functions indicates that, if v exceeds a certain value, the value of the two functions will be less than the smallest discrete value acceptable by the system, determined by the number of fractional bits available.

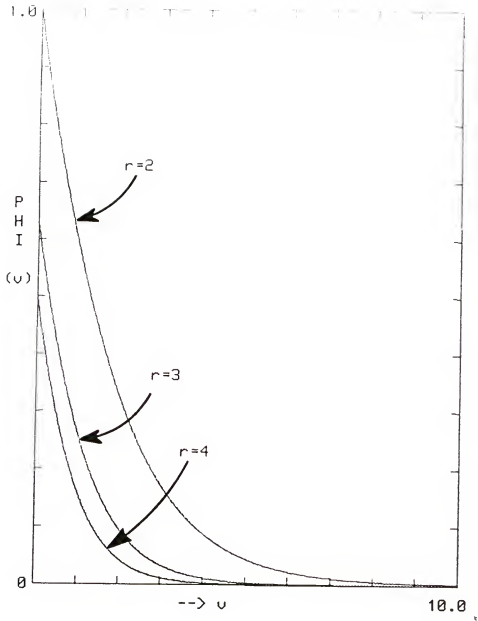


FIGURE 5.1
Logarithmic Addition Mapping: $\Phi(v)$ versus v

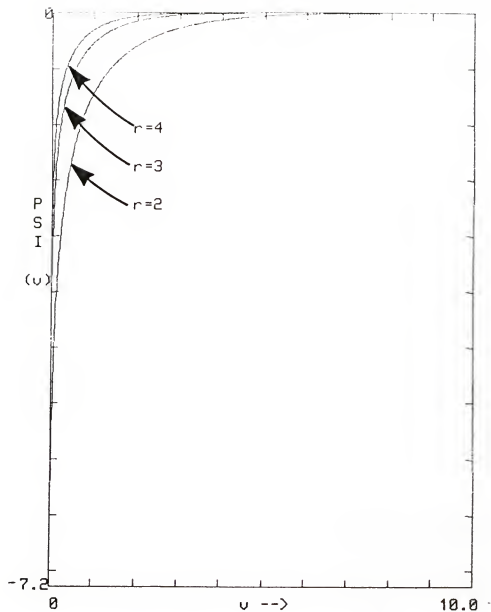


FIGURE 5.2
Logarithmic Subtraction Mapping: $\Psi(v)$ versus v

This observation was first made by Kingsbury and Rayner [Kin71]. This value will be referred to as "essential zero", and denoted Z_A for the addition and Z_S for the subtraction. Again, in Figures 5.1 and 5.2 it can be observed that as r increases, the values of Z_A and Z_S are reduced. By using a proof similar to the one used for the base optimization in Chapter Three, for a given r , these two variables can be proven to depend only on the dynamic range one wants to cover and the LNS fractional wordlength. They can be computed as follows:

$$\begin{aligned}
 \Phi(Z_A) = \lg(1+r^{-Z_A}) &\leq 2^{-(F+1)} && \rightarrow \\
 1 + r^{-Z_A} &\leq r(2^{-F-1}) && \rightarrow \\
 r^{-Z_A} &\leq r(2^{-F-1}) - 1 && \rightarrow \\
 Z_A &\geq -\lg\left(r(2^{-F-1}) - 1\right) && (5.1)
 \end{aligned}$$

and

$$\begin{aligned}
 \Psi(Z_S) = \lg(1-r^{-Z_S}) &\geq -2^{-F+1} && \rightarrow \\
 1 - r^{-Z_S} &\leq r(-2^{-F-1}) && \rightarrow \\
 r^{-Z_S} &\geq 1 - r(-2^{-F-1}) && \rightarrow \\
 Z_S &\leq -\lg\left(1 - r(-2^{-F-1})\right) && (5.2)
 \end{aligned}$$

By using the equations 5.1 and 5.2, some indicative values

of Z_A and Z_S , with respect to certain values of the base r and the fractional wordlength F , are tabulated in Table 5.1. The code for generation of the entries of Table 5.1 is listed in Appendix G. It can be observed that for values of $F > 6$ and $r = 2$, the values of the essential zeros converge at about $F + 1.52$. So, even though the dynamic range V extends, say, to 2^{128} , requiring 7 integer bits, the number of fractional bits being limited to, say 12, would reduce the real need down to 4 bits, since the essential zeros would only be 13.528. This would automatically reduce the storage requirements by a factor of 8.

5.2 Optimization of Base

Another kind of optimization can be achieved, if the wordlength and the dynamic range to be covered are given. The base can be determined in such a way that the essential zero will be a power of two. This will allow for no unused space in the ROM, since for any number F of inputs, there will be no multiple values of zeros stored in the table. For an essential zero of $V = 2^V$ and a fractional wordlength of F bits, this value of optimal base can be computed as follows:

$$1r \left(1 + r^{-2^V} \right) \leq 2^{-F-1} \quad \text{or}$$

$$1 + r^{-2^V} \leq r^{(2^{-F-1})} \quad \text{or} \quad r^{-2^V} \leq r^{(2^{-F-1})} - 1 \quad \text{or}$$

TABLE 5.1
Essential Zeros for Various Values of r and F

r	F	z_A	z_S
2	5	6.520947	6.536572
2	6	7.524858	7.532671
2	7	8.526813	8.530719
2	8	9.527790	9.529743
2	9	10.528278	10.529255
2	10	11.528522	11.529011
2	11	12.528644	12.528888
2	12	13.528705	13.528827
2	13	14.528736	13.528797
e	12	9.010852	9.010974
e	8	6.237347	6.239301
4	12	6.264322	6.264444
1.293	12	40.354559	40.354681
1.088	12	136.158753	136.158875

$$r^{(2^{-F}-1)} - r^{-2^v} \geq 1 \quad (5.3)$$

By solving inequality (5.3) numerically, the values of base as a function of v and F were computed and are shown in Table 5.2. It can be seen that all of them are between one and two, thus offering one more justification for choosing the base to be greater than unity. A less than unity choice has been made by various researchers [Lan85] for data compression purposes.

5.3 Other Table Reduction Techniques

Frey and Taylor [Fre85] proposed an algorithmic method for memory table reduction is presented. It is based on the "staircase" shape of the functions Φ and Ψ and results in at least two bits of savings when applied in addition to the essential zero technique. All of the above methods, plus the precision enhancing methods to be presented in the next chapters, show a feasible way to the realization of a logarithmic processor having a precision of more than 20 bits.

TABLE 5.2
Optimal Bases for Given F and V ($=2^V$)

F	v	r
3	3	1.460
5	3	1.674
4	3	1.562
4	4	1.293
8	7	1.067
12	7	1.088

CHAPTER SIX ADAPTIVE RADIX PROCESSOR

Many researchers have devoted their efforts to find precision enhancing methods for LNS, because of the restricted wordlength of the fast memory tables and in addition to the aforementioned table reduction techniques. Fruit of this effort, for example, is the practical hardware interpolation method offered by Taylor [Tay83]. In the following section, two more methods will be presented. One of them, called the Adaptive Radix Processor (ARP), is a "random memory access" method, while the other, hereafter called as Associative Memory Processor (AMP), is a "sequential access" one.

6.1 Adaptive Radix Processor Principle

The Adaptive Radix Processor partitions the table look-up address space, in order to minimize the finite wordlength effects. It is premised on the fact that the address $v = |x - y|$, generated for the addition or subtraction memory look-ups, is likely to have some, (i), leading zeros. Such an address can be shifted by a shift register to the left by i bits. Next it can be presented to the table(s) performing the mappings required for logarithmic addition or subtraction respectively

$$\phi_i(v_i) = \text{lr} \left(1 + r_i^{-v_i} \right) \quad \text{and} \quad \psi_i(v_i) = \text{lr} \left(1 - r_i^{-v_i} \right) \quad (6.1)$$

$$\text{with} \quad v_i = 2^i v \quad \text{and} \quad r_i = r^{2^{-i}}.$$

The final result of the operation will then be the same as if the normal logarithmic addition or subtraction was looked-up, and ready to be used directly in any further computations. The only difference is that now i more bits of information have been "compressed" into the table space, reducing thus the total entropy of the system. The way that the ARP processor is architected for implementation of the logarithmic add/subtract cycle is presented in Figure 6.1. It is the processor's ability to adaptively select the radix during run time, that contributed to its naming. Next, an error analysis that aspires to theoretically justify the new processor will be presented.

6.2 Error Analysis for the ARP

Multiplication is error free in LNS. Therefore, an error analysis for the ARP needs only to examine the addition part of the processor.

The general error model used for the analysis is shown in Figure 6.2. There, the real LNS exponents of the two numbers to be added are denoted as x and y . Their machine versions are x^* and y^* respectively. They are associated through the errors a_1 and a_2 , which were

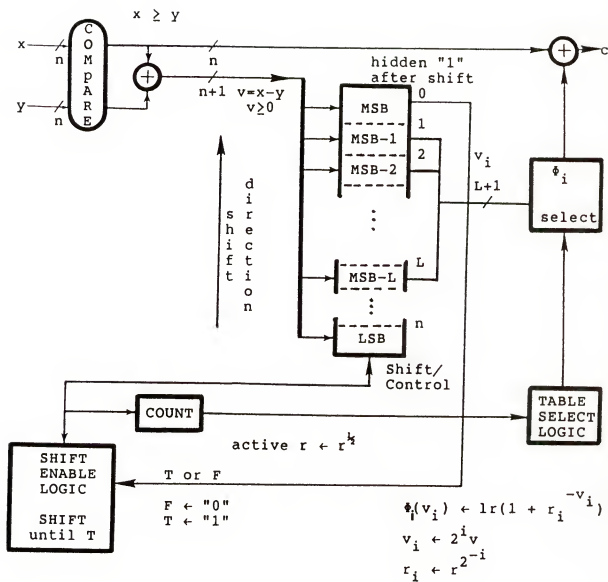


FIGURE 6.1
Add Cycle Implementing the ARP Policy

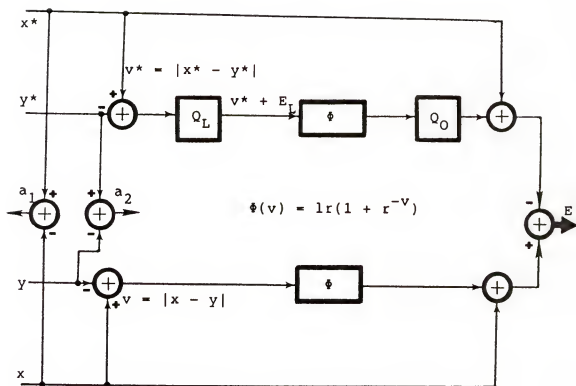


FIGURE 6.2
Error Model for Logarithmic Addition

examined in Chapter Four and found to have trapezoidal probability density functions. For the analysis presented here, they are assumed to be independent random variables. In an ideal environment (not in the processor), v would be presented to a table offering as outputs the homomorphism $\Phi(v) = \text{lr}(1 + r^{-v})$ and the LNS exponent of the result of the addition would be $x + \Phi(v)$. The machine version of v (v^*) is presented to a memory table as a number having a precision of 2^{-F+I} , where F is the number of fractional bits used for the output of the FLP to LNS conversion table $\Theta(m_x) = \text{lr}(m_x)$ and I is the number of (integer only) bits used for the representation of x' (as in the FLP representation $X = m_x r^{x'}$). Here v^* is a random variable having a triangular p.d.f. as shown in Figure 6.3, where Z is the already familiar "essential zero," discussed in Chapter Five. The triangular shape of $f_{v^*}(v^*)$ is explained by the fact that v is the absolute difference of two uniformly distributed numbers [Pap65]. It suggests that it is likely for v^* to have some, say L , leading zeros, supplying thus the basis for the ARP architecture.

To be able to check the effects of the shiftings suggested by the ARP, the quantizer Q_L is provided in the error model, introducing thus an error

$$E_L \in U\left[-\frac{q_L}{2}, \frac{q_L}{2}\right] \quad \text{with} \quad q_L = 2^{-F+I-L} \quad (6.2)$$

Of course, the error model should provide and for a

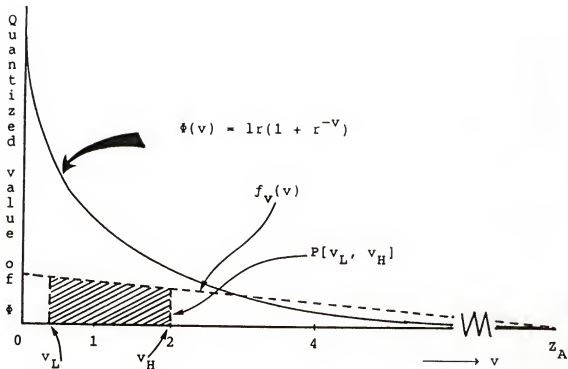


FIGURE 6.3
P.d.f. for $v = |x - y|$

quantizer Q_0 at the output of the "ideal" table $\Phi(v)$, to take care of the finite precision effects introduced by the table wordlength. This introduces the error

$$E_0 \in U\left[-\frac{q_0}{2}, \frac{q_0}{2}\right] \quad \text{with } q_0 = 2^{-F} \quad (6.3)$$

Finally the error E at the output of the LNS adder is given as

$$\begin{aligned} E &= \lg(r^x + r^y) - \lg(r^{x*} + r^{y*-E_L}) - E_0 \quad \text{or} \\ E &= \lg(r^x + r^y) - \lg(r^{x-a_1} + r^{y-a_1-E_L}) - E_0 \quad \text{or} \\ E &= \lg[r^x(1+r^{y-x})] - \lg\left[r^x\left(r^{-a_1+r^{y-x-a_1-E_L}}\right)\right] - E_0 \quad (6.4) \end{aligned}$$

Without loss of generality, it can be assumed that $x > y$. Then for $v = |x - y|$ one finds

$$\begin{aligned} E &= x + \lg(1 + r^{-v}) - x - \lg\left(r^{-a_1} + r^{-v-a_2-E_L}\right) - E_0 \quad \text{or} \\ E &= \lg(1 + r^{-v}) - \lg\left(r^{-a_1} + r^{-v-a_2-E_L}\right) - E_0 \quad (6.5) \end{aligned}$$

Examining equation 6.5 one can observe that the p.d.f. for E cannot be directly computed, because of the presence of the random variable v , whose joint density function with any of the other random variables (a_1 , a_2 , E_0 , E_L) is unknown or difficult to obtain. To bypass this problem, one can assume that v is fixed (in other words consider it

to be a deterministic variable) and calculate analytical expressions for the mean and variance of E . Next, one can integrate over an admissible range for v , divide the result by the range metric and finally multiply it with the probability that v belongs to that specific range. If v_L and v_H are the low and high limits of a range for v , then the probability that v belongs to that domain is calculated to be

$$P[v_L, v_H] = \frac{2}{Z} \left[v_H - v_L - \frac{v_H^2 - v_L^2}{2Z} \right] \quad (6.6)$$

Since E_L is always bounded by $q_L/2$ and a_1, a_2 are bounded by $\lg(1 + q) + q_1/2$, with $q = 2^{-F}$ and $q_1 = q/2$, as proved in Chapter Four, one can find a condition under which E_L will be in general greater than a_1 and a_2 . This condition is

$$2^{-F+I-L-1} > \lg(1 + 2^{-F}) + 2^{-F-2} \quad \text{or} \\ L < I - F - 1 - \lg[\lg(1 + 2^{-F}) + 2^{-F-2}] \quad (6.7)$$

Operating at the limits of the available technology, that is $F \approx 12$, equation (6.7) is reduced to the condition $L < I - 2$. Based on this assumption, equation (6.5) can be rewritten as

$$E = \lg(1 + r^{-v}) - \lg(r^{-v-E_L}) - E_0 \quad (6.8)$$

The necessity of taking into account the errors, resulting from the encoding from FLP to LNS, can be made

apparent from the limit values for E in Table 6.1. Several values for E are shown there for different values of v , I , and L . In all cases, the number of fractional bits $F = 12$ and the base is always taken as $r = 2$. The variable E_{com} stands for the comprehensive error after the encoding effects are taken into account, whereas E stands for the error, without any encoding effects being considered. Several claims can be validated by observing this table. Firstly, the larger the shift the less is the error. Secondly, for a fixed value of v and constant difference of I and L , the error remains the same. Thirdly, a small value of I means a small error too. Finally, in all cases, E is less than E_{com} .

The analysis can now proceed, since the random variable P , where $P = \text{lr}(1 + r^{-v-E_L})$, can be analyzed as a function of another random variable (E_L). Solving for E_L

$$E_L = -v - \text{lr}(r^P - 1) \quad (6.9)$$

$$\begin{aligned} \text{and} \quad \frac{\partial P}{\partial E_L} &= \frac{1}{\log r} \frac{1}{1 + r^{-v-E_L}} \frac{\partial(1 + r^{-v-E_L})}{\partial E_L} \\ &= \frac{1}{\log r} \frac{1}{1 + r^{-v-E_L}} (-1) \log r \, r^{-v-E_L} \\ &= - \frac{r^{-v-E_L}}{1 + r^{-v-E_L}} \end{aligned} \quad (6.10)$$

$$\text{and} \quad \left| \frac{\partial P}{\partial E_L} \right|_{E_L} = \frac{r^P - 1}{1 + r^P - 1} = \frac{r^P - 1}{r^P} \quad (6.11)$$

TABLE 6.1
Limits of Error at the Output of the Logarithmic Adder.

v	I	L	$E \times 10^{-6}$		$E_{com} \times 10^{-6}$	
			-low	high	-low	high
.5	5	4	223	223	637	636
.5	5	3	324	324	738	738
.5	5	2	527	526	940	940
.5	5	1	931	931	1345	1345
.5	5	0	1741	1739	2155	2152
.5	8	7	223	223	637	636
.5	8	5	527	526	940	940
.5	8	3	1741	1739	2155	2152
.5	10	3	6615	6574	7028	6987
.5	10	2	13148	12984	13562	13977
.1	5	0	2009	2006	2422	2419
.1	10	7	594	593	1007	1007
.1	10	3	7685	7643	8098	8056
.01	4	16	244	244	657	657
.01	3	16	365	365	779	779
.01	0	16	2070	2067	2483	2480

E_{com} is with and E is without any errors, resulting from encoding from other arithmetic systems, being considered.

so that

$$f_P(P) = \frac{f_{E_L}(E_L)}{\left| \frac{\partial P}{\partial E_L} \right|_{E_L}} = \frac{r^P}{q_L(r^P - 1)} \quad \text{for } L_1 \leq P \leq L_2 \quad (6.12)$$

$$\text{with } L_1 = \ln \left(1 + r^{-v} - \frac{q_L}{2} \right) \quad (6.13)$$

$$\text{and } L_2 = \ln \left(1 + r^{-v} + \frac{q_L}{2} \right) \quad (6.14)$$

If $M = \ln(1 + r^{-v-E_L}) + E_0$ or $M = P + E_0$, then

$$f_M(M) = \int_{-\infty}^{\infty} f_P(P) f_{E_0}(M-P) dP = \frac{1}{q_0} \int_{M-\frac{q_0}{2}}^{M+\frac{q_0}{2}} f_P(P) dP \quad (6.15)$$

To compute the value of the above integral, precise knowledge of the relative positions of M and the limits of the domain of the variable P must be at hand. To facilitate the understanding of the algebra involved, the unnecessary complexity can be avoided by defining

$$\begin{aligned} \Lambda &= \frac{1}{q_L q_0}, & M^{\pm} &= M \pm \frac{q_0}{2} \quad \text{and} \\ K^{\pm} &= \ln(r^{M^{\pm}} - 1), & K &= \ln(1 + r^{-v}) \quad \text{and} \\ N^{\pm} &= \ln \left(r^{K-E_{\pm} \frac{q_0}{2}} - 1 \right), & L_i^{\pm} &= L_i \pm \frac{q_0}{2}; \quad i = 1, 2 \end{aligned} \quad (6.16)$$

Case i) $L_2^- > L_1^+$ or $q_0 < L_2 - L_1$ (6.17)

Then M is moving along the axis



and $f_M(M)$ is found to be given by

$$f_M(M) = \begin{cases} \Lambda \int_{L_1^-}^{M^+} \frac{r^P}{r^P - 1} dP & \text{for } L_1^- \leq M \leq L_1^+ \\ \Lambda \int_{M^-}^{M^+} \frac{r^P}{r^P - 1} dP & \text{for } L_1^+ \leq M \leq L_2^- \\ \Lambda \int_{M^-}^{L_2^-} \frac{r^P}{r^P - 1} dP & \text{for } L_2^- \leq M \leq L_2^+ \end{cases} \quad (6.18)$$

or

$$f_M(M) = \begin{cases} \Lambda \left[K^+ - \ln(r^{L_1} - 1) \right] & \text{for } L_1^- \leq M \leq L_1^+ \\ \Lambda [K^+ - K^-] & \text{for } L_1^+ \leq M \leq L_2^- \\ \Lambda \left[\ln(r^{L_2} - 1) - K^- \right] & \text{for } L_2^- \leq M \leq L_2^+ \end{cases} \quad (6.19)$$

or

$$f_M(M) = \begin{cases} \Lambda \left[K^+ + v + \frac{q_L}{2} \right] & \text{for } L_1^- \leq M \leq L_1^+ \\ \Lambda [K^+ - K^-] & \text{for } L_1^+ \leq M \leq L_2^- \\ \Lambda \left[-v + \frac{q_L}{2} - K^- \right] & \text{for } L_2^- \leq M \leq L_2^+ \end{cases} \quad (6.20)$$

Case ii) $L_2^- < L_1^+$ or $q_0 > L_2 - L_1$ (6.21)

Then M is moving along the axis



and $f_M(M)$ is found to be given by

$$f_M(M) = \begin{cases} \Lambda \int_{L_1^-}^{M^+} \frac{r^P}{r^P - 1} dP & \text{for } L_1^- \leq M \leq L_2^- \\ \Lambda \int_{L_1^-}^{L_2} \frac{r^P}{r^P - 1} dP & \text{for } L_2^- \leq M \leq L_1^+ \\ \Lambda \int_{M^-}^{L_2} \frac{r^P}{r^P - 1} dP & \text{for } L_1^+ \leq M \leq L_2^+ \end{cases} \quad (6.22)$$

or

$$f_M(M) = \begin{cases} \Lambda \left[K^+ + v + \frac{q_L}{2} \right] & \text{for } L_1^- \leq M \leq L_2^- \\ \Lambda \left[-v + \frac{q_L}{2} + v + \frac{q_L}{2} \right] = \frac{1}{q_0} & \text{for } L_2^- \leq M \leq L_1^+ \\ \Lambda \left[-v + \frac{q_L}{2} - K^- \right] & \text{for } L_1^+ \leq M \leq L_2^+ \end{cases} \quad (6.23)$$

Finally, the error at the output of the logarithmic adder is given by

$$E = \ln(1 + r^{-v}) - M = K - M \quad (6.24)$$

Solving for M , $\Rightarrow M = K - E$ and $\left| \frac{\partial E}{\partial M} \right|_M = |-1| = 1$. Then

$$f_E(E) = \frac{f_M(E)}{\left| \frac{\partial E}{\partial M} \right|_M} = f_M(K-E) \quad (6.25)$$

Again, for the same two cases as for M, expressions for $f_E(E)$ are computed as

Case i) If $q_0 < L_2 - L_1$, then

$$f_E(E) = \begin{cases} \Lambda \left[N^+ + v + \frac{q_L}{2} \right] & \text{for } K - L_1^- \leq E \leq K - L_1^+ \\ \Lambda \left[N^+ - N^- \right] & \text{for } K - L_2^+ \leq E \leq K - L_1^- \\ \Lambda \left[-v + \frac{q_L}{2} - N^- \right] & \text{for } K - L_2^- \leq E \leq K - L_2^+ \end{cases} \quad (6.26)$$

or

$$f_E(E) = \begin{cases} \Lambda \left[-v + \frac{q_L}{2} - N^- \right] & \text{for } K - L_2^- \leq E \leq K - L_2^+ \\ \Lambda \left[N^+ - N^- \right] & \text{for } K - L_2^+ \leq E \leq K - L_1^- \\ \Lambda \left[N^+ + v + \frac{q_L}{2} \right] & \text{for } K - L_1^- \leq E \leq K - L_1^+ \end{cases} \quad (6.27)$$

It can be observed that

$$\left\{ \begin{array}{l} a = K - L_2^- \\ b = K - L_2^+ \\ c = K - L_1^- \\ d = K - L_1^+ \end{array} \right\} \rightarrow \left\{ \begin{array}{l} f_E(a) = 0 \\ f_E(b) = \Lambda \left[\frac{q_L}{2} - v - 1r(r^{L_2 - q_0 - 1}) \right] \\ f_E(c) = \Lambda \left[\frac{q_L}{2} + v - 1r(r^{L_1 + q_0 - 1}) \right] \\ f_E(d) = 0 \end{array} \right\} \quad (6.28)$$

Then, the complicated expression (6.27) can be very accurately approximated by the trapezoidal shaped

function, shown in Figure 6.4 and expressed as

$$f_E(E) = \begin{cases} (E - a) \frac{TOP}{b - a} = (E - a) \frac{TOP}{q_0} & \text{for } a \leq E \leq b \\ TOP & \text{for } b \leq E \leq c \\ (E - d) \frac{TOP}{c - d} = (d - E) \frac{TOP}{q_0} & \text{for } c \leq E \leq d \end{cases} \quad (6.29)$$

where TOP can be computed by requiring that

$\int_{-\infty}^{\infty} f_E(E) dE = 1$, as

$$\frac{TOP}{q_0} \int_a^b (E - a) dE + TOP \int_b^c dE - \frac{TOP}{q_0} \int_c^d (E - d) dE = 1$$

$$\text{or} \quad TOP = \frac{1}{L_2 - L_1} \quad (6.30)$$

As explained earlier, the mean and variance of the error E are necessary for the final justifications and they are computed as

Mean and Variance of E for case i)

The mean of E is given by

$$\begin{aligned} M_1[E] &= \int_{-\infty}^{\infty} f_E(E) E dE = \\ &= \frac{TOP}{q_0} \int_a^b (E - a) E dE + TOP \int_b^c E dE - \frac{TOP}{q_0} \int_c^d (E - d) E dE \\ &= \frac{TOP}{q_0} \left[\frac{b^3}{3} - \frac{a^3}{3} - \frac{ab^2}{2} + \frac{a^3}{2} - \frac{d^3}{3} + \frac{c^3}{3} + \frac{d^3}{2} - \frac{dc^2}{2} \right] + \\ &\quad TOP \left[\frac{c^2}{2} - \frac{b^2}{2} \right] \end{aligned} \quad (6.31)$$

and the variance of E is

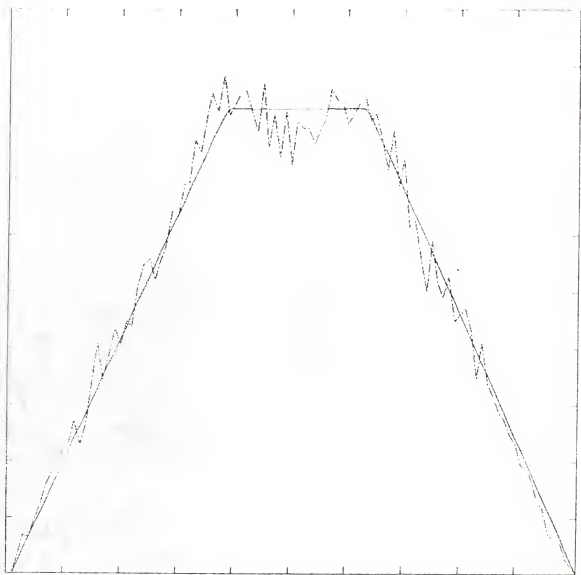


FIGURE 6.4
Shape of $f_E(E)$ for Case i) of LNS Addition

$$\begin{aligned}
D_1[E] &= \int_{-\infty}^{\infty} f_E(E) E^2 dE - M_1^2[E] = \\
&= \frac{TOP}{q_0} \int_a^b (E-a) E^2 dE + TOP \int_b^c E^2 dE - \frac{TOP}{q_0} \int_c^d (E-d) E^2 dE - M_1^2[E] \\
&= \frac{TOP}{q_0} \left[\frac{b^4}{4} - \frac{a^4}{4} - \frac{ab^3}{3} + \frac{a^4}{3} - \frac{d^4}{4} + \frac{c^4}{4} + \frac{d^4}{3} - \frac{dc^3}{3} \right] + \\
&\quad TOP \left[\frac{c^3}{3} - \frac{b^3}{3} \right] - M_1^2[E] \quad (6.32)
\end{aligned}$$

Case ii) If $q_0 > L_2 - L_1$, then

$$f_E(E) = \begin{cases} \Lambda \left[-v + \frac{q_L}{2} - N^- \right] & \text{for } K - L_2^- \leq E \leq K - L_1^- \\ \frac{1}{q_0} & \text{for } K - L_1^- \leq E \leq K - L_2^+ \\ \Lambda \left[N^+ + v + \frac{q_L}{2} \right] & \text{for } K - L_2^+ \leq E \leq K - L_1^+ \end{cases} \quad (6.33)$$

It can be observed that

$$\left\{ \begin{array}{l} i = K - L_2^- \\ j = K - L_1^- \\ k = K - L_2^+ \\ l = K - L_1^+ \end{array} \right\} \rightarrow \left\{ \begin{array}{l} f_E(i) = 0 \\ f_E(j) = \frac{1}{q_0} \\ f_E(k) = \frac{1}{q_0} \\ f_E(l) = 0 \end{array} \right\} \quad (6.34)$$

Then, the complicated expression (6.33) can be very accurately approximated by the trapezoidal-like function,

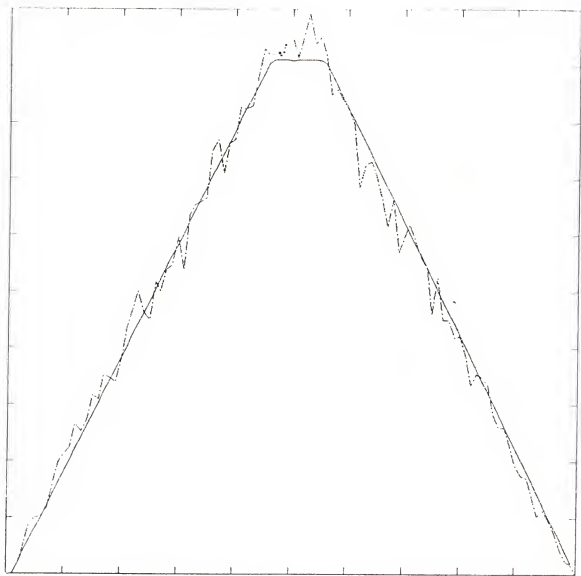


FIGURE 6.5
Shape of $f_E(E)$ for Case ii) of LNS Addition

shown in Figure 6.5 and expressed as

$$f_E(E) = \begin{cases} (E - i) \frac{1}{q_0(j-1)} & \text{for } i \leq E \leq j \\ \frac{1}{q_0} & \text{for } j \leq E \leq k \\ (E - 1) \frac{1}{q_0(k-1)} & \text{for } k \leq E \leq 1 \end{cases} \quad (6.35)$$

or

$$f_E(E) = \begin{cases} (E - i) \frac{TOP}{q_0} & \text{for } i \leq E \leq j \\ \frac{1}{q_0} & \text{for } j \leq E \leq k \\ (1 - E) \frac{TOP}{q_0} & \text{for } k \leq E \leq 1 \end{cases} \quad (6.36)$$

where TOP has the same value as in case i.

Mean and Variance of E for case ii)

The mean of E is given by

$$\begin{aligned} M_2[E] &= \int_{-\infty}^{\infty} f_E(E)E \, dE = \\ &= \frac{TOP}{q_0} \int_i^j (E-i)E \, dE + TOP \int_j^k E \, dE - \frac{TOP}{q_0} \int_k^1 (E-1)E \, dE \\ &= \frac{TOP}{q_0} \left[\frac{j^3}{3} - \frac{i^3}{3} - \frac{ij^2}{2} + \frac{i^3}{2} - \frac{1^3}{3} + \frac{k^3}{3} + \frac{1^3}{2} - \frac{1k^2}{2} \right] + \\ &\quad TOP \left[\frac{k^2}{2} - \frac{1^2}{2} \right] \end{aligned} \quad (6.37)$$

and the variance of E is

$$D_2[E] = \int_{-\infty}^{\infty} f_E(E)E^2 \, dE - M_2^2[E] =$$

$$\begin{aligned}
&= \frac{\text{TOP}}{q_0} \int_i^j (E-i)E^2 \, dE + \text{TOP} \int_j^k E^2 \, dE - \frac{\text{TOP}}{q_0} \int_k^1 (E-d)E^2 \, dE - M_2^2[E] \\
&= \frac{\text{TOP}}{q_0} \left[\frac{j^4}{4} - \frac{i^4}{4} - \frac{ij^3}{3} + \frac{i^4}{3} - \frac{1^4}{4} + \frac{k^4}{4} + \frac{1^4}{3} - \frac{1k^3}{3} \right] + \\
&\quad \text{TOP} \left[\frac{k^3}{3} - \frac{j^3}{3} \right] - M_2^2[E] \tag{6.38}
\end{aligned}$$

In Figures 6.4 and 6.5 the solid lines represent both the complete analytical forms of $f_E(E)$, given by the expressions (6.27) and (6.33) for the two cases examined, as well as approximate analytical forms, described by the equations (6.29) and (6.33) respectively. The two curves are shown to be absolutely identical, in both cases. The interrupted lines represent experimental curves generated for a variety of values for the variables v , L , I , F . Because of the similarity of these curves, only one example is shown for each of the two distinct cases. The coincidence with the theoretical curves is shown to be almost identical. For both figures the sample size was taken to be 30,000. F was chosen to be 12, v was .5 and I was 5. For Figure 6.4, L was 3, whereas for Figure 6.5, L was taken to be 5. These curves were produced after dividing the values of the histograms, that were generated for E , into the histograms' steps and the sample sizes. The code for the theoretical form of $f_E(E)$ is listed in Appendix H, while the plot of its theoretical approximation is generated by the code listed in Appendix I. Appendix J contains the code for the computer simulated error of the logarithmic adder.

Using the equations (6.31)-(6.32) and (6.37)-(6.38), according to the case, a computer search was performed, based on the procedure described after equation (6.5). The results permitted the sophisticated partitioning of the range of addresses into T adjoining regions (i.e.; compact cover) which are the address subranges for T dedicated look-up tables. The criterion of choice was the one of minimal overall error variance, described by the above formulas. The results for optimal three and two-level decompositions of the address space, for various values of the base r , F , I , and Z are tabulated in Table 6.2. The blank spaces in this table indicate that condition (6.7) was taken into account and large multibit shifts were prevented from taking place, in order to have control over the results with the derived analytical formulas. The listing of the code used to generate the entries for Table 6.2 can be found in Appendix K. All of the above analysis holds true, not only when $L \neq 0$. In other words not only when the ARP idea has been implemented, but also in the special case, where no shifting of the address v takes place. The only modification that has to be made to the formulas already derived is to set $L = 0$. On the other hand, condition (6.7) does not allow for large multibit shifts, if the capability for quantitative prediction of the error behavior of the LNS processor is desirable. But large shifts require a significant increase in the number of additional tables to be used for the $\phi_i(v)$ mappings.

TABLE 6.2
Experimental Values For the Error Variance
for Two and Three-Level Partitioning
and Without Using ARP at all.

r	F	Z	I	Error Variance $\times 10^{-10}$		
				3-level	2-level	No Break
2	12	8	8	4370	11100	197000
2	12	8	7	1130	2820	49300
2	12	8	6	3180	744	12300
2	12	8	5		223	3130
2	9	8	7	72000	180000	3150000
2	9	8	6	20300	47500	791000
2	9	8	5		14200	2000
2	9	8	4		17300	52400
2	9	16	3		14500	15400

This can be costly and contributes to the complexity of the system. Also, the analysis showed that, by imitating a signal amplifier, it is sufficient to shift the addresses (which are very close to zero) even only by 1 bit in order to obtain significant reduction in the error variance. This suggests that at most 2 additional tables (a total of three) will usually bring the error variance down to acceptable levels.

6.3 Presorting of Operands

The preceding error analysis reveals that the maximum accuracy gain exists whenever the absolute difference of the two LNS operands to be added or subtracted is very small so that additional information can be packed into the data through the shifting procedure. In the general ARP-LNS case the assumption was made that the two operands were uniformly distributed and in no degree correlated. This was the basis for the assumption of a triangular shape for $f_v(v)$. However, if the luxury (for modest investment in sorting hardware and the latency it suggests) can be afforded by the specific application, then $f_v(v) \rightarrow \delta_k(0)$ (Kronecker delta function). This implies that, for a maximum allowable address-shift of L bits, the error variance, would be reduced by a factor of 2^{-2L} , thus resulting in major dividends, when compared to non-ARP designs.

6.4 Signed-digit ARP-LNS

The ARP design problem acquires a very interesting dimension when alternative number systems become prospective employees. Such an instance is when the redundant signed-digit (SD) system and especially the canonical one is used for representing the binary LNS exponents. A brief review of the canonical system follows.

6.4.1 Signed-Digit Number Systems

Following Hwang [Hwa79], SD numbers are formally defined in terms of a radix r , and digits which can assume the following $2\alpha+1$ values

$$V_r = \{-\alpha, \dots, -1, 0, 1, \dots, \alpha\}$$

where the maximum digit magnitude α must be within the region

$$\left\lceil \frac{r-1}{2} \right\rceil \leq \alpha \leq r-1$$

The original motivation of using SD number system was to eliminate the need for carry propagation chains in an addition/subtraction task. To break the carry chain it is necessary to make the lower bound on α tighter as

$$\left\lceil \frac{r+1}{2} \right\rceil \leq \alpha \leq r-1$$

The number of nonzero digits in an n -digit SD vector, say,

$$Y = (y_{n-1} \cdot \dots \cdot y_1 y_0 \cdot y_{-1} y_{-k})_r \text{ with value } Y_v = \sum_{i=-k}^{n-1} y_i r^i,$$

is called the weight $\omega(n, Y_v)$. In general the weight of a

binary n -digit SD vector is defined below with

$$|y_i| = 1 \text{ if } y_i \neq 0$$

$$\omega(n, Y_v) = \sum_{i=0}^{n-1} |y_i|$$

The SD vector with the minimal weight is called a minimal SD representation with respect to given values of n and Y_v . A minimal SD vector $D = D_{n-1} \dots D_1 D_0$ that contains no adjacent nonzero digits is called a canonical signed-digit vector. Reitwiesner [Rei60] showed that there exists a "unique" canonical SD form D for any digital number with a fixed value α and a fixed vector length n , provided the product of the two leftmost digits in D does not equal one, that is

$$D_{n-1} \times D_{n-2} \neq 1$$

This property can be always satisfied by imposing an additional digit $D_n = 0$ to the left end of the vector D .

A procedure to transform a $(n+1)$ -digit binary vector $B = B_n B_{n-1} \dots B_1 B_0$ with $B_n = 0$ and $B_i \in \{0, 1\}$ for $0 \leq i \leq n-1$, to a canonical SD vector $D = D_n D_{n-1} \dots D_1 D_0$ with $D_n = 0$ and $D_i \in \{\bar{1}, 0, 1\}$ such that both vectors represent the same value is given below, where the carry vector is symbolized as C

Step 1. Start with LSB of B . Set $i=0$, $C_0=0$.

Step 2. Examine B_{i+1} , B_i and C_i to generate

$$C_{i+1} = B_{i+1}B_i + B_iC_i + B_{i+1}C_i$$

Step 3. Generate $D_i = B_i + C_i - 2C_{i+1}$

Step 4. Increment index i by one. Go to step 2 if

$i < n$. Stop otherwise.

The conversion from canonical to binary can be done by subtracting the digit vector that results if only the negative digits of D are considered and setting the rest of them to zero, from the one that results if only the positive digits are considered.

6.4.2 Signed-digit Addition

The addition or subtraction of two SD numbers X and Y is considered for the following discussion. The numbers involved are assumed to be $(n+k)$ -digit SD numbers

$$X = (x_{n-1} \cdot \cdot \cdot x_1 x_0 \cdot x_{-1} x_{-k})_r \quad \text{and}$$

$$Y = (y_{n-1} \cdot \cdot \cdot y_1 y_0 \cdot y_{-1} y_{-k})_r.$$

The i th sum digit of the resulting sum

$$S = (s_{n-1} \cdot \cdot \cdot s_1 s_0 \cdot s_{-1} s_{-k})_r = X + Y$$

is s_i and t_i is the transfer digit (carry) from the $(i-1)$ th digital position. The difference of the transit digit from the regular carry or borrow digits is that, unlike these, it may assume negative values. An "interim sum" digit w_i is generated in intermediate stages.

According to Avizienis [Avi61], a totally parallel addition or subtraction, restricts signed-digit representations to radices $r \geq 2$. Furthermore, at least $r+2$ values are required for the sum digits s_i and requirements for parallel subtraction would increase the required number of subtrahend digit values to $r+3$ for even

radices. However, the digit addition rules may be modified to allow for propagation of the transfer digit over two digital positions to the left. If this two-transfer addition is allowed, the radix $r=2$ may be used and only $r+1$ values are required for the sum digit. Two-transfer addition is executed in the three successive steps shown below (the adder diagram from Avizienis [Avi61] is shown in Figure 6.6 and an addition example is given in Figure 6.7)

$$\begin{aligned} 1) \quad x'_i + y'_i &= rt'_{i-1} + w'_i \\ 2) \quad w'_i + t'_i &= rt''_{i-1} + w''_i \\ 3) \quad s'_i &= w''_i + t''_i \end{aligned}$$

where

$$t'_{i-1} = \begin{cases} 1 & \text{if } x_i + y_i \geq 1 \\ 0 & \text{if } x_i + y_i = 0 \\ -1 & \text{if } x_i + y_i \leq -1 \end{cases} ; \quad t''_{i-1} = \begin{cases} 1 & \text{if } w'_i + t'_i = 2 \\ 0 & \text{otherwise} \\ -1 & \text{if } w'_i + t'_i = -2 \end{cases}$$

and the digits take values:

$$x_i, y_i, s_i, t'_i, t''_i, w'_i, w''_i \in \{\bar{1}, 0, 1\}.$$

From the Figure 6.6 it can be seen that there is a carry propagation over two successive digit positions. Avizienis [Avi61] concluded that, in general, the lower limit on the required redundancy of one digit is a function of the number of digital positions over which a signal is allowed to propagate. If no redundancy exists and each sum digit assumes only r values, a sum digit s_i

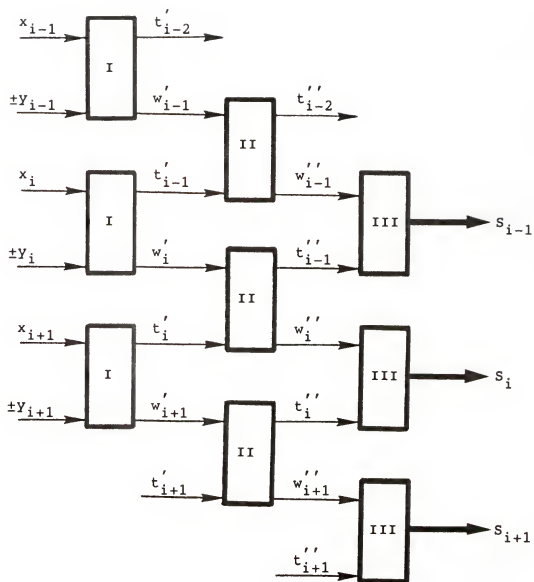


FIGURE 6.6
Basic Computation Kernel for a Two-Transfer Addition

$$x = (\bar{1}.01001)_2 = (-.71875)_{10}$$

$$y = (0.100\bar{1}0)_2 = (.4375)_{10}$$

$$s = (0.0\bar{1}00\bar{1})_2 = (-.28125)_{10}$$

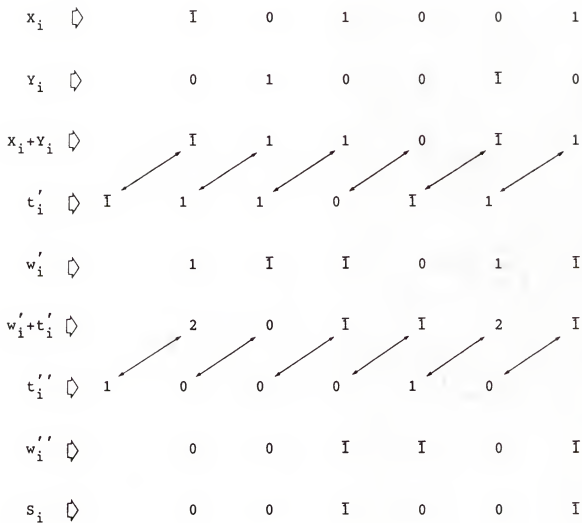


FIGURE 6.7
Two-Transfer Addition Example

is a function of all the addend digits x_i and the augend digits y_i to its right, i.e., $s_i = f(x_i, y_i, x_{i+1}, \dots, x_m, y_m)$. If the sum digit assumes $r+1$ values, then $s_i = f(x_i, y_i, x_{i+1}, y_{i+1}, x_{i+2}, y_{i+2})$ and the operation is the two-transfer addition. If each sum digit assumes $r+2$ values or more, then $s_i = f(x_i, y_i, x_{i+1}, y_{i+1})$ and with the restriction $r \geq 3$, the addition is totally parallel. It is clear that the price to be paid for totally parallel addition is the required (by the redundancy) additional storage.

6.4.3 Canonical ARP-LNS

The canonical system can be employed for representing the LNS exponents. Since all of the operations, including multiplications and divisions, are based on additions or subtractions and table look-ups, it is important to be able to perform these operations as fast as possible. The time requirements for any operation are greatly reduced into the time, which is required for the execution of the three steps that the two-transfer addition calls for. If the basic computational kernel, shown in Figure 6.6, is used in a pipeline mode, then the memory table input address calculation may start with the most significant digit first. According to the ARP LNS design, for a L-partition problem, L memory tables will be candidates for access. The generation of the first $L-1$ digits of the address v will be enough for the memory chip selection,

which can be done even before the calculation of the address is completed. This introduces additional savings, that write off the kernel computation time. In a non-pipelined mode the computation of the address v will be carried out in an almost parallel fashion and the whole operation will only require the kernel computation time. Since LNS multiplication and division are reduced to normal addition, this presents the side advantage that the execution time for these operations will remain basically independent of the LNS wordlength and the timing considerations for a VLSI design are thus greatly minimized. Expansions of the wordlength based on technological advances in memory design and construction, effecting the design of addition and subtraction, will not effect the timing considerations for multiplication or division. At the expense of a little more storage hardware, accommodating the prepended MSB digit (always zero) and the redundancy introduced by the canonical digit representation, a very fast and modular engine has evolved.

CHAPTER SEVEN ASSOCIATIVE MEMORY PROCESSOR

7.1 Associative Memory Processor Principle

The established way to access a memory table requires to store all items in the storage medium where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Several search algorithms have been developed, trying to minimize the number of accesses, while searching for an item in memory. The time required to find the item can be reduced considerably, if the stored data can be identified for access by their content, generally some specified subfield, rather than by their address. Memory units accessed by content are generically classified as associative memories or content addressable memories (CAM) [Koh78]. Because of its organization, the associative memory (AM) is uniquely suited for parallel searches by data association. Moreover, searches can be done on a entire word or on certain fields within a word.

The idea of using AM to support the LNS look-up task is examined in this dissertation. It can be implemented as follows: The N -bit address field is partitioned into a set of adjoint subfields. The subfield data of v are presented to a section of the associative memory, which provides a partial answer to the value of $\Phi(v)$ or $\Psi(v)$. As a means for illustration of the method, suppose that the N -bit field of the input address v is split into two subfields. The most significant block (MSB) consists of N_1 bits of data, with $(N_1 - I) = F_1$ fractional bits (I are the integer bits of v), while the least significant one (LSB) consists of $N_2 (= N - N_1)$ bits. The mapping offered by the first module of AM as a response to MSB may belong into one of two possible classifications

1. Unambiguous (denoted U)
2. Ambiguous (denoted A)

There will be an ambiguous result whenever N_1 bits of input are not enough to determine a unique mapping. Otherwise the result will be unambiguous. Consider the data offered in Table 7.1, where the discrete independent variables v are mapped into discrete outputs through $\Phi(v)$. It is taken that the final output consists of $F=5$ bits, the input consists of $N=7$ bits and the dynamic range extends from 0.4 to 0.52. It can be observed that if $N_1=4$, there are two A states: 0110 and 1000, resulting into more than one outputs, and one U state: 0110, resulting only in output 11010. If $N_1=5$, then again there

TABLE 7.1
 Ambiguous and Unambiguous States Generated by
 the Associative Memory Modules of the AMP.

Decimal v	Decimal $\Phi(v)$	Binary v	Binary $\Phi(v)$	States
0.398437 0.406250 0.414062 0.421875 0.429687	0.812500 0.812500 0.812500 0.812500 0.812500	0110011 0110100 0110101 0110110 0110111	11010 11010 11010 11010 11010	U
0.437500	0.812500	0111000	11010	A1 ₁
0.445312 0.453125 0.460937 0.468750 0.476562 0.484375 0.492187	0.781250 0.781250 0.781250 0.781250 0.781250 0.781250 0.781250	0111001 0111010 0111011 0111100 0111101 0111110 0111111	11001 11001 11001 11001 11001 11001 11001	A1 ₂
0.500000 0.507812	0.781250 0.781250	1000000 1000001	11001 11001	A2 ₁
0.515625	0.750000	1000010	11000	A2 ₂

N=7, F=5, N₁=4, Range: 0.4 → 0.52

are two A states: 01110 and 10000 (with fewer members though), but more U states, and so on. The ambiguity will have to be resolved by using the LSB of input information. This, in addition to some information regarding the A states encountered in the first module, will be input to the second AM module, which will produce the final answer.

In general, there could be up to K R-bit-wide associative memory modules, which will accept up to R_i bits of data each. This data field would consist of N_i bits of information from the present subfield of input and S_{i-1} bits of information generated by the previous AM module as suggested in Figure 7.1. The table will export an F-bit output U, or if that is not feasible for the specific module, a S-bit wide information word for the next module. Each table will be able to perform an ordinal magnitude comparison over a $S_{i-1} + N_i = R_i \leq R$ -bit field and map the result into a F-bit output field. If the decision by the associative memory is that the input represents a U state, then the F-bit output field is filled with the precomputed rounded value of $\Phi(v)$. A detected A condition will cause a (dis)abling bit to be set so as to clear the outputs of any of the AM modules, which are laid to its right. Of course, the critical issue is how to partition N into K N_i -bit fields, so that

$$N = \sum N_i ; \quad i = 1, 2, \dots, K$$

For $i > 1$ the individual N_i will have to share the table address space with the S_{i-1} -bit word arriving from the

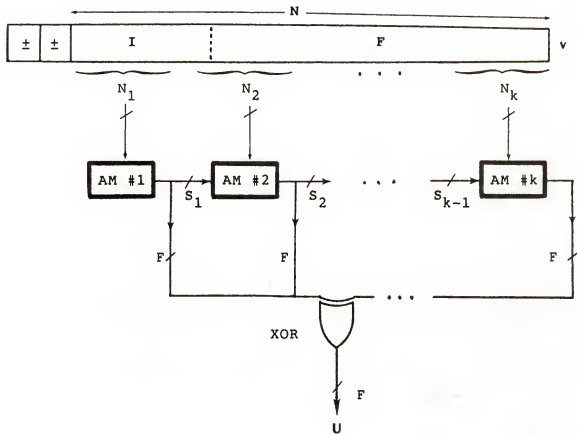


FIGURE 7.1
General LNS Associative Memory Processor Principle

table's left neighbor. Of course, since S_i carries the information needed for distinguishing between the A states, the number of bits committed to S_i is determined by the total number of A states detected at that level. It will be equal to $\lceil \lg(\text{number of A states}) \rceil$.

7.2 Optimization

The discrete and rounded versions of $\Phi(v)$ and $\Psi(v)$, not being homomorphisms, do not allow for a prediction of the number of U or A classes, which may evolve from a given partition. Factors that influence the cardinality of these sets are the base r , the number of associative modules K and the address space of these modules, determined by R . The degree of their influence can be estimated and relative optimization can be achieved only through a numerical study using dynamic programming principles. The two design attributes that are examined from an optimization standpoint in this dissertation are the speed and the complexity. Considered to be given, were

- R : maximum address space for a semiconductor memory with cycle time t_R
- N : unsigned LNS wordlength

Although not a general rule, technology suggests that if $R_1 < R_2$, then $t_{R_1} < t_{R_2}$. In a pipelined architecture, the optimization target would be to minimize the largest necessary address space of memory table. In other words, the function $\phi_1(R) = \min \left\{ \max \{N_i + S_{i-1}\} \right\}$ for $i = 1, \dots, K$

should be minimized. If the goal is to minimize latency (non pipelined throughput), then the function

$$\phi_2(R) = \min \left\{ \sum_{i=1}^K t_{R_i} \right\} \quad \text{should be optimized. The values of}$$

t_{R_i} can be obtained from data books, or modeled as $t_{R_i} = c \lg(R_i)$, where c is a proportionality constant and $\lg(R_i)$ reflects the addressing overhead. Finally, if complexity is the issue then the number K of AM modules should be minimized. Since, in practice, K would be a relatively small integer with R ranging from 6 to 14 bits, an iterative search procedure could be used, to optimize the design with respect to the chosen criteria of optimality. Summarizing, the side constraints imposed to the optimization search should be the following:

$$\begin{aligned} 1. \quad O_1(K, R) &= \sum_{i=1}^K N_i \geq N \\ 2. \quad O_2(K, R) &= N_i + S_{i-1} = R_i \leq R \\ 3. \quad O_3(K, R) &= S_{i-1} \leq F; \quad i > 1, \quad S_0 = 0 \end{aligned} \quad (7.1)$$

where again, N_i is the blocksize of the i th subfield, R is the maximum allowable address space for the AM modules, F is the output wordlength (only fractional bits), K is the population of AM modules and S_i is the number of bits exported by the i th module, determined by the number of A classes ($S_i = F$, if the module offers a U answer).

7.2.1 Latency Estimation

Based on the above optimization goals, several experiments were carried out. A numerical search was performed to locate the minimum value for $\phi_1(R)$, for various values of N , F , and r . The code used is listed in Appendix L and some of the obtained values are tabulated in Table 7.2. Analysis of these results showed that for specified dynamic range (determined by I), fractional wordlength, and number of AM modules, the minimum value of $\phi_1(R)$ was always $N - K + 1$. For example, for $K=4$ and $N=18$, it was never found to be less than 15. This minimum value was also found to be the average value, which means that all of the AM modules would present the same amount of (rather big) slowness. Investing one more bit for the slowest table, some of the tables could be constructed with a minimum input address length of up to three bits, but this tradeoff does not result in any substantial savings, while on the other hand it could complicate the design of the processor by eliminating the existing uniformity.

7.2.2 Tree Structure Experiments

A binary tree structure spawning both A and U states can be employed to implement the associative mappings. For example, the data found in Table 7.1 would require a tree like the one shown in Figure 7.2 (for $N_1 = 4$). Not all of

TABLE 7.2
Optimal Values of $\phi_1(R)$ for AM Processor Design
for Varying Values of N, K, F and $r=2$.

	i	N_i	S_{i-1}	$\phi_1(R)$
N=18 F=14 K=4	1	15	0	15
	2	1	14	
	3	1	14	
	4	0	13	
N=12 F=9 K=3	1	10	0	10
	2	1	8	
	3	1	7	
N=8 F=5 K=3	1	1	0	6
	2	5	1	
	3	2	4	
N=18 F=14 K=3	1	2	0	16
	2	14	2	
	3	2	14	
N=18 F=15 K=3	1	15	0	16
	2	2	14	
	3	1	13	

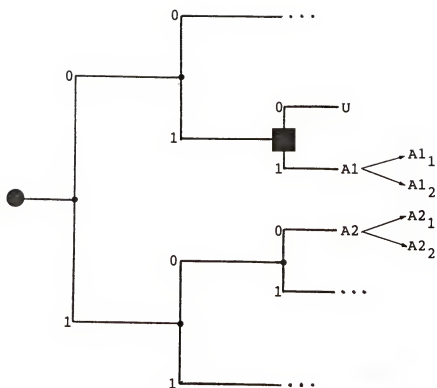


FIGURE 7.2
Tree-Structured AM module of AMP (see Table 7.1)

the tree nodes are necessarily occupied by switches. Depending on the ambiguity or not of the resulting states, a tree node may become a leaf node even though strictly it doesn't belong to the last level of the tree. If this is the case, the need for a switch is eliminated. An attempt was also made to employ a bit-serial architecture by constructing the whole memory table via a tree structure. Experimentation was performed regarding the amount of hardware (in this case the switches necessary to implement the corresponding trees). The results were obtained via the code listed in Appendix M and are depicted in Figure 7.3. It can be seen that the number of switches increases exponentially with the fractional wordlength. The demand in switches becomes really huge after F becomes greater than 13 bits.

7.2.3 Effect of Base on Clustering

Different requirements for the input address space of the memory table are also the result of a different choice for the base of the LNS processor. Again, experimentation was performed to determine the effect of different radices on clustering of the input address space. The code used is again the one listed in Appendix L and some of the results are shown in Table 7.3. It can be observed that different radices do not result into reduced input address space, unless there is a substantial decrease in the number of bits used for the output mapping representation and,

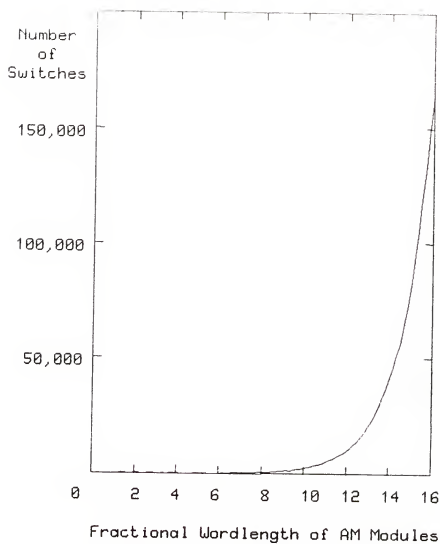


FIGURE 7.3
Number of Switches versus F for Tree-Structured
AM Modules

TABLE 7.3
Effect of Radix Variation on the Input Address Space
for the AM Processor

r	i	N_i	S_{i-1}	$\phi_1(R)$
2	1 2 3	2 5 3	0 1 3	6
3	1 2 3	2 6 2	0 1 5	7
3	1 2 3	5 3 2	0 1 4	6
4	1 2 3	3 4 3	0 1 2	5
4	1 2 3	2 5 3	0 1 4	7
0.85	1 2 3	1 1 8	0 1 2	10

$N=10$, $F=3$, $I=4$, $K=3$, Range: $0 \rightarrow 16$.

afterall, there is never more than one bit of savings. This is not worth the extra hardware burden imposed by the deviation from the choice of $r = 2$ and the reduction that is suffered by the precision of the LNS processor itself.

7.3 Comparison of AMP and ARP

The above analysis, together with the one presented in Chapter Six, proves the ARP LNS design to be superior to the AMP one, not only in terms of speed (there is no latency for the ARP), but also in terms of actual memory cells, required to achieve a certain amount of precision.

CHAPTER EIGHT COMPLEXITY OF OPERATIONS FOR LNS PROCESSORS

8.1 Lower Bounds of Arithmetic Operations

The major advantage of LNS is its ability to perform fast multiplication and division. Therefore, in addition to the need for achieving the lowest possible count of operations, which are required for a given algorithm, it seems plausible to try to replace a number of additions or subtractions involved in a specific algorithm with multiplications or divisions. An extensive survey of literature, related to this area of optimization [Kir77, Mor73, Mot55, Ost54, Pan63, Pan66, Pan83, Tod55, Win68, Win70] revealed many interesting results, which will be presented below along with some efforts of ours in the area.

The design and analysis of arithmetic algorithms for Discrete Fourier Transforms (DFT), Convolution of Vectors (CV), Cyclic Convolution of Vectors (CCV), and Matrix Multiplication (MM) have been examined by Pan [Pan83]. He analyzed the arithmetic and logical complexity of such algorithms and focused on the classes of (bi)linear algorithms defined below. If I, J, K, i, j, k are nonnegative integers, μ is a matrix, $(\mu)_{ij}$ is the entry of μ lying in

row i and column j , P is a field of constant coefficients, X, Y, Z are vectors of indeterminates, $x_i = (X)_i$, $y_j = (Y)_j$, $z_k = (Z)_k$, $x_i = y_j = z_k = 0$ (unless $0 \leq i < I$, $0 \leq j < J$, $0 \leq k < K$), and f_{ij} , $f_{ijk} \in P$ for all i, j, k , then, following Pan, the linear and bilinear arithmetic computational problems are defined as

Definition 8.1 A linear problem is a set of linear forms

$$l_k(Y) = \sum_{j=0}^{J-1} f_{jk} y_j \quad \text{for } k = 0, 1, \dots, K-1 \quad (8.1)$$

Definition 8.2 A bilinear problem is a set of bilinear forms

$$b_k(X, Y) = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} f_{ijk} x_i y_j \quad \text{for } k = 0, 1, \dots, K-1 \quad (8.2)$$

Also a linear problem can be equivalently represented by a bilinear form $b(Y, Z)$, or by a matrix, μ such that

$$b(Y, Z) = \sum_{k=0}^{K-1} l_k(Y) z_k, \quad (\mu)_{jk} = f_{jk} \quad \text{for all } j, k \quad (8.3)$$

Some examples of linear or bilinear problems of particular interest to Digital Signal Processing are given below, where m, n are nonnegative integers.

Problem 8.1 DFT. Discrete Fourier Transform at $n+1$ points.

$$b(Y, Z) = \sum_{k=0}^n z_k \sum_{j=0}^n \omega^{jk} y_j \quad (\mu)_{jk} = \omega^{jk}$$

where ω is a $(n+1)$ -root of unity.

Problem 8.2 CV. Convolution of two vectors or polynomial multiplication.

$$t(X, Y, Z) = \sum_{k=0}^{m+n} z_k \sum_{j=0}^k x_{k-j} y_j \quad (\mu(X))_{jk} = x_{k-j}$$

where $x_i = 0$ if $i < 0$ or $i > m$, $y_j = 0$ if $j > n$

Problem 8.3 CCV. Cyclic Convolution of vectors or multiplication of two n th degree polynomials in (λ) , modulo $\lambda^{n+1} - 1$.

$$t(X, Y, Z) = \sum_{k=0}^n z_k \left(\sum_{j=0}^k x_{k-j} y_j + \sum_{j=0}^{\bar{k}} x_{\bar{k}-j} y_j \right)$$

$$(\mu(X))_{jk} = x_{k-j} \quad \text{if } j \leq k \quad (\mu(X))_{jk} = x_{\bar{k}-j} \quad \text{otherwise}$$

Here $j, k = 0, 1, \dots, n$, $\bar{k} = k + n + 1$.

Problem 8.4 MM. Matrix Multiplication ($I=J=K=n^2$). Now the vectors X, Y, Z , are represented as the matrices X, Y, Z , so that $(X)_i = (X)_{\alpha\beta}$, $(Y)_j = (Y)_{\beta\gamma}$, $(Z)_k = (Z)_{\gamma\alpha}$, $i = \alpha n + \beta$, $j = \beta n + \gamma$, $k = \gamma n + \alpha$, and $\alpha, \beta, \gamma, = 0, 1, \dots, n-1$

$$t(X, Y, Z) = \sum_{\alpha, \beta, \gamma=0}^{n-1} x_{\alpha, \beta} y_{\beta, \gamma} z_{\gamma, \alpha} = \text{Tr}(X, Y, Z),$$

$$\begin{aligned} (\mu(X))_{jk} &= (X)_{\alpha\beta} & \text{if } j = \beta n + \gamma n, k = \gamma n + \alpha \text{ for some } \alpha, \beta, \gamma \\ (\mu(X))_{jk} &= 0 & \text{otherwise} \end{aligned}$$

The (bi)linear algorithms are natural ones, easy to implement and analyze. They form the basis of the fastest known DSP algorithms and all other DSP algorithms can be turned into (bi)linear ones with the total number of operations remaining constant. For the analysis of the arithmetic complexity of linear and bilinear algorithms, it is technically convenient to estimate separately the total number of additions/subtractions $C(\pm)$, nonscalar multiplications $C(*)$, and scalar multiplications C_{sc} , which are involved in the algorithm. Assume that the determinates $(X)_i$ are replaced by their values taken from P . Then, of course, all bilinear algorithms turn into linear ones and their $C(\pm)$ does not increase. An arbitrary (bi)linear algorithm can be written as a sequence of linear forms in the input variables Y . Then Pan suggests the following lower bounds for $C(\pm)$: $C(\pm) + do(D(A)) = O(n \lg n)$ for Problems 8.1-8.3 and $O(n^2 \lg n)$ for Problem 8.4, with "disorder" $do(D(A))$ defined later in this section.

By using straightforward substitution techniques, Ostrowski [Ost54] showed that the lower bounds on $C(\pm)$ range between K and $K+Q$, where K and Q are the total numbers of input variables and linearly independent outputs of the algorithm. This means that n additive operations are necessary for polynomial algorithms of degree n . These bounds cannot become the lowest possible unless severe restrictions are imposed on the model of

computations, but then no constant can be allowed to be greater than unity [Mor73].

Pan associates acyclic digraphs $D = D(A)$ with the (bi)linear algorithms A . Their vertices have outdegrees 2 and 0 and represent the $+$ operations of A and the input variables of A respectively. Then $C(\pm)$ is the number of vertices of D that have outdegrees 2. In order to estimate that number, all vertices of D can be partitioned into $p(D)$ levels ($p(D)$ being the "profundity" of D), whose cardinalities are bounded from below by r (the number of linearly independent outputs of the problem. Then $p(D)r \leq C(\pm)$. If the desirable partition does not exist (due to some deficiencies in the structure of the algorithm A represented by the digraph $D(A)$), then $C(\pm)$ is augmented to include measures of the "irregularity" of D ($ir(D)$), and "disorder" (quantitative measure of the asynchronicity and structural deficiency of the algorithms according to others) of D ($do(D)$). The terms $ir(D)$ and $do(D)$ can be alternatively interpreted as measures for the additional storage required, since the deficiencies they represent are a result of precedence problems ([Tay83b] p.214). Pan shows that both deficiencies can be corrected by transforming D . This is accomplished by joining some appropriate paths to it. It is shown that a total of $ir(D)+do(D)$ new vertices suffice to produce a new digraph with no irregularity or disorder and with the same profundity of the old one. The additive complexity has

become now $C(\frac{+}{-}) + \text{ir}(D) + \text{do}(D)$. In the case of the four DSP problems mentioned above, $p(D)r$ is asymptotically proportional to $(K+Q)\lg(K+Q)$. Finally, an estimate of $C(\frac{+}{-})$ from below is offered for these problems. If $r(P(X))$ is the rank of an arbitrary polynomial $P(X)$ and $r(m)$ the rank of any minor m of the matrix $\mu(X)$, that defines a bilinear computational problem, then it is shown that $C(\frac{+}{-}) \geq \lg(r(m))$.

A notion of rank or independence for arbitrary sets of rational functions is developed [Kir77]. This rank bounds from below the number of additions and subtractions required for all straight line algorithms which compute those functions. This permits the uniform derivation of the lowest bounds, which are known for a number of familiar sets of rational functions. These bounds are reported below, where D denotes now an arbitrary integral domain, F is the quotient field of D , P is an arbitrarily large pool of distinct indeterminates, A is any rational algorithm over $(F(P), \text{DUP})$, and $C(\frac{+}{-})$ is the number of addition/subtraction steps in A :

A1. If A computes the expression $a_1 + a_2 + \dots + a_n$ then $C(\frac{+}{-}) \geq n-1$.

A2. If A computes the expression

$$\frac{a_1 + a_2 + \dots + a_n}{a_{n+1} + a_{n+2} + \dots + a_{n+m}} \quad \text{then } C(\frac{+}{-}) \geq n+m-2.$$

A3. If A computes the pair of expressions $a_1 a_3 - a_2 a_4$

and $a_1a_4 + a_2a_3$ (the real and imaginary parts of the complex product $(a_1+a_3i)(a_2+a_4i)$) then $C(\pm) \geq 2$.

A4. If A computes the general rational function

$$\frac{a_n x^n + a_{n-1} x^{n-1} + \dots + a_0}{b_m x^m + b_{m-1} x^{m-1} + \dots + b_0} \quad \text{then } C(\pm) \geq n+m.$$

A5. If A computes the matrix-vector product

$$M \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}, \quad \text{then } C(\pm) \geq N^*(M) - t.$$

where M is a $t \times n$ matrix over D, and $N^*(M)$ denotes the number of columns of M that are not identically zero.

A6. If A computes the matrix product AX , then $C(\pm) \geq (m+p-1)(n-1)$, where A and X are $m \times n$ and $n \times p$ matrices respectively.

In particular this gives addition/subtraction lower bounds of $m(n-1)$ for the product of an $m \times n$ matrix with an n -vector, and $2n^2 - 3n + 1$ for the product of two $n \times n$ matrices.

A7. If A computes the matrix-vector product

$$X \begin{bmatrix} a_n \\ \vdots \\ a_1 \\ a_0 \end{bmatrix}, \quad \text{with} \quad X = \begin{bmatrix} x_1^n & x_1^{n-1} & \dots & x_1 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \dots & x_m & 1 \end{bmatrix}$$

then $C(\pm) \geq n+m-1$.

This means that the evaluation of an n th degree polynomial at m arbitrary points requires at least $n + m - 1$ additions/subtractions.

A8. If A computes the set P_1, \dots, P_t , with

$$P_i = \sum_{j=0}^{n(i)} a_{ij} x^j; \quad i = 1, \dots, t \quad \text{then} \quad C(\pm) \geq \sum_{i=1}^t n(i).$$

A9. If A computes the expression

$$\sum_{i=0}^n \sum_{j=0}^n a_{ij} x^i y_j \quad \text{then} \quad C(\pm) \geq (n+1)^2 - 1.$$

8.2 Preconditioning of Coefficients

Motzkin [Mot53] introduced the notion of preconditioning of the coefficients. He showed that if operations, depending only on the a_i in the course of computing the expression, $\sum_{i=0}^n a_i x^i$, are not counted, then only about $n/2$ multiplications are necessary in order to compute that expression. The obvious application of this result is when the same polynomial has to be computed

at many different points x . This is the case with the inversion of a square matrix or the solution of a system of n linear equations. It should be mentioned that the substitution algorithms offered by Winograd reduce the number of multiplications by increasing the number of additions so that the total number of basic operations remains almost constant. This is not desirable in a LNS environment. There, the opposite is the goal. A technique to minimize the multiplications (by increasing the number of additions) was first offered by Todd [Tod55]. The reverse line of reasoning will be followed in order to obtain LNS gains in the following example:

Assume that the polynomial

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

has to be evaluated. This polynomial can be written as

$$\left((b_0x) + b_1\right)\left((b_2x + b_3)x + a_4\right)$$

By equating the coefficients the following relations can be found between the a 's and b 's:

$$a_4 = b_0b_2$$

$$a_3 = b_0b_3$$

$$a_2 = b_1b_2 + b_0b_4$$

$$a_1 = b_1b_3$$

$$a_0 = b_1b_4$$

Then for $a_2 = a_4 \frac{a_1}{a_3} + a_0 \frac{a_3}{a_1}$ and for any choice of b_0 it

can be found that

$$b_1 = \frac{a_1 a_2 - a_3 a_0}{a_1 a_4} b_0 \quad \text{and}$$

$$b_2 = \frac{a_4}{b_0}, \quad b_3 = \frac{a_1}{b_1}, \quad b_4 = \frac{a_0}{b_1}.$$

This formulation results in a scheme, which requires only three additions, while the number of multiplications becomes five (an increase of one with regard to the optimal Horner's rule). Notice that the forming of the factor x^2 does not require more than a mere shift in LNS, and that for $b_0=1$ the preconditioning results in the savings of one addition and one multiplication. If this polynomial evaluation has to be performed for many different x 's then the preconditioning of the coefficients in the described way, which requires five multiplications and one addition, will prove to be worthwhile and will contribute to savings in speed in the long run. Similar addition count reduction schemes can be found for other polynomials as well, but this will have to be part of the preparation for the execution of the specific application.

8.3 Simultaneous Computation of Polynomials

The idea of preconditioning can be also applied for the case of computing together the values of several fixed polynomials at the same point (real or complex) for several values of the indeterminate. Such is the case,

for example, of approximate computation with increasing degrees of accuracy. Such "interlacing" of preconditioning schemes has been reported two decades ago [Pan66]. The lower bound for $C(\frac{1}{2})$ was given as $N - s$, and the lower bound for $C(*)$ as $\frac{N-s+2}{2}$, where N is the total number of variable and independent coefficients and s is the number of polynomials to be simultaneously computed. Some algorithms for the construction of such schemes have been suggested and analyzed in the same work.

CHAPTER NINE

LNS DSP APPLICATIONS

The basic LNS unit, whether mechanized as an ARP or not (AMP designs will not be considered furthermore since they are inferior to the ARP ones), will be capable of performing a limited number of important DSP algorithms in a Reduced Instruction Set Computer (or RISC-like) environment. In either case, throughput will be limited by the speed of its components. For comparative purposes, several commercially available floating-point processors of short wordlength are compared in Table 9.1. Using the Shottky TTL (2500LS) design as a reference and parts from the same semiconductor family plus a 4K 35ns ROM for parameters, the estimated performance of the LNS and LNS-ARP is reported in Table 9.2. It can be noted that for add/subtract intensive operations, conventional floating-point and LNS processor latency are comparable. However in real multiply intensive environment, a significant advantage is enjoyed by the LNS. This advantage becomes more obvious when commanding the complex arithmetic.

The entire procedure for the development of applications can be systematized by using the following steps:

TABLE 9.1
Comparison of Commercially Available Short-Wordlength
Floating-point Processors

Operation	INTEL 8087* (ms)	WEITEK WTL 1032-5 32-bits $f_{c/k}=100\text{ns}$ (ns)	AMD-MSI 2500LS ** 16-bit mantissa (ns)	Sky 2910-based 32-bit (ns)
ADD	.019	910	74-111 derived	5300
SUBTRACT	.140	910	74-111 derived	8500
MULTIPLY	.021	910	160 derived	5500
DIVIDE	.159	N/A	N/A	15000
SQRT	.159	N/A	320	66370

* Based on 10^6 sequentially called executions
on single precision data

** Fast increment/decrement = 10ns

Fixed point : add = 37ns, multiply = 150ns

TABLE 9.2
Throughput Estimates for Various Basic DSP Operations
Using Conventional and LNS processors

Operation		A Conventional ¹	B LNS ² (ns)	A/B latency (ns)
R	ADD, SUB	111	2(37)+45=119	0.933
	MULT	160	37	4.324
	DIV	320(est)	37	8.649
E	SQRT ³	320	10(S/R)	32.000
	SQUARE	160	10(S/R)	16.000
A	RMS ⁴	320+160L +111(L-1)+320= 271L + 529	37L+119(L-1) +10+37= 156L - 58	~1.737
L	FIR ⁵	111(L-1)+160L= 271L-111	119(L-1)+37L= 156L-119	~1.737
C O M P L E X	ADD, SUB	222	239	0.929
	MULT	4(160)+222= 1262	4(37)+239= 386	3.269
	ABS	2(160)+111+ 320 = 751	2(10)+119+10= 149	5.040

¹Based on survey tables

²Based on a 45ns HMOS ROM/RAM

³Model: 2 × FLP multiplication delay

⁴RMS calculation: $\left(\sum_{i=1}^L x_i^2 \right)^{1/2} / L$

⁵FIR computation: $\sum_{i=1}^L a_i x_i$

- Step 1) Precompile the application algorithm to maximize the multiplication/addition tradeoff. Here, techniques introduced in Chapter Eight, including rearrangement of equations and preconditioning of coefficients can be used.
- Step 2) Extract maximum parallelism in computations, through the use of the mathematical models for precedence relations.
- Step 3) Compile and execute a simulation program to verify and accumulate statistics and resource requirements. Such a simulation should determine, among others, the percentage of calls of specific memory tables, corresponding to certain subranges of an ARP design.
- Step 4) Based on the information gathered in step 3, configure the system, including the optimal resources and execute the application.

The LNS has already been shown to be a viable tool in implementing FFTs and linear shift-invariant filters. Several basic operations required for digital filter applications are suggested in Table 9.2, where an Lth order FIR and sums of products algorithms are compared. However, an attempt to showcase some of the more interesting non-obvious DSP applications of the LNS will be made below.

9.1 Multiple Input Adder

Many DSP operations are intrinsically sums of products statements. Therefore, there may be some advantage in combining derived partial product terms with a tree-adder. The LNS components developed to this point accept a maximum of two operands. Since add times are dominant in LNS, it is desirable to explore potentially faster multi-operand adders. Consider a three operand adder configured to add $S = A + B + C$. Then

$$s \leftarrow \text{lr}(1 + r^{b-a} + r^{c-a}) = a + \Xi((b-a), (c-a)) \quad (9.1)$$

$$\text{with } \Xi(x, y) = \text{lr}(1 + r^x + r^y)$$

The lookup address space would have to be sufficiently large so as to accommodate the concatenated address $b-a$, $c-a$. An alternative expression for s is the following:

$$s \leftarrow a + \text{lr}(1 + r^{b-a} + \Phi(b-c)) \quad (9.2)$$

or

$$s \leftarrow a + \Phi(a - b - \Phi(b-c)) \quad (9.3)$$

Such an adder can be realized easily in LNS, since it only requires the use of the very same table used for regular two-operand addition. Some extra routing hardware has to be provided, especially to cover the case of a subtraction being a sub-operation. The basic architecture of such a three-operand addition is shown in Figure 9.1 for a non-pipelined operation. If the hardware has to be optimized,

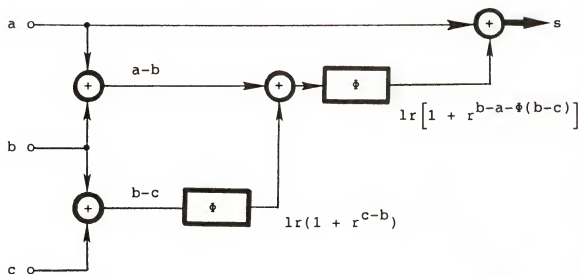


FIGURE 9.1
Three-Operand LNS Adder

simple rerouting suffices to make multiple use of the same hardware elements such as memory tables. The advantage of this architecture over the two-operand addition is that one magnitude comparison time is saved. The price for that lies in the extra bit for the address space normally required for the address space of the second table, but even this drawback could be alleviated by appropriately adjusting the dynamic range of the LNS, to exploit the fact that the output of the Φ table is always bounded by unity.

9.2 CORDIC Replacement

By use of a controlled set of a prescribed sequence of conditional additions or subtractions the CORDIC equations can be used to approximately solve either set of the following equations:

$$\begin{aligned} Y' &= K (Y \cos \lambda + X \sin \lambda) \\ X' &= K (X \cos \lambda - Y \sin \lambda) \end{aligned} \quad (9.4)$$

or

$$\begin{aligned} R &= K (X^2 + Y^2)^{1/2} \\ \Theta &= \tan^{-1}(Y/X) \end{aligned} \quad (9.5)$$

where K is an invariable constant. Though the concept of CORDIC arithmetic is said to be quite old [Vol59], its implementations and applications continue to evolve especially in areas like computer graphics and analysis [Dag59, Hav80]. The acronym comes from Volder's COordinate

Rotations Digital Computer, developed in 1959 for air navigation and control instrumentation. In 1971 Walther [Wal71] generalized with elegance the mathematics of CORDIC, showing that the implementation of a wide range of transcendental functions can be fully represented by a single set of iterative operations. All operations are based on the execution (in a multiplier-free digital system) of either

$$x_{i+1} = x_i \pm Y_i 2^{-i} \quad (9.6)$$

or

$$x_{(i+1),2} = (1 + \gamma 2^{-i}) x_{(i+1),1}$$

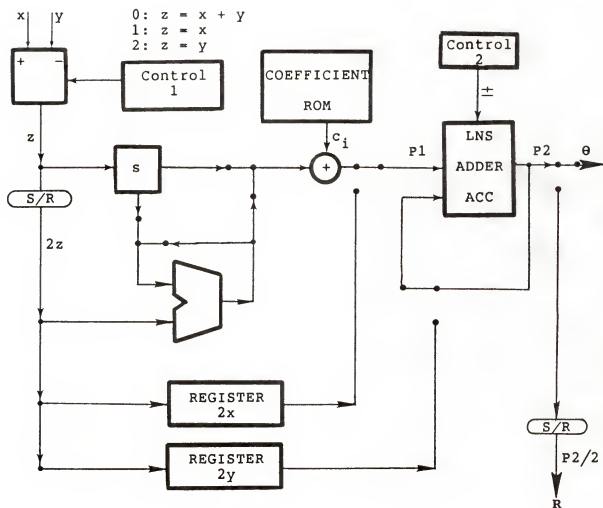
While the first of these equations represents the regular (Rotation and Vectoring) CORDIC operations, the second forces the scale factors of circular and hyperbolic functions to unity. ROM based instructions govern the selection of either the first or the second equation. The architecture suggested by (9.6) is multiplier-free and uses only elementary shifts or adds. It implies a variable execution time ranging from a very short delay to one of the order of F shift and add delays, where F is the number of bits of the dataword.

For a system with a fast multiplier, such as the LNS, the direct calculation for the variables is proved to be faster and more accurate and regular from a data flow aspect. The generation of polar from rectangular coordinates (Vectoring) is considered. Then the equations

(9.5) have to be computed (for $K = 1$). The efficient generation of R in the LNS has been already reported in Table 9.2, while the production of Θ can be accomplished through the use of a series expansion of the form

$$\Theta = \tan^{-1}(S) = \begin{cases} \frac{\pi}{2} - \frac{1}{S} + \frac{S^3}{3} - \frac{S^5}{5} + \frac{S^7}{7} + \dots ; S > 1 \\ S - \frac{S^3}{3} + \frac{S^5}{5} - \frac{S^7}{7} + \dots ; S^2 < 1 \\ -\frac{\pi}{2} - \frac{1}{S} + \frac{S^3}{3} - \frac{S^5}{5} + \frac{S^7}{7} + \dots ; S < -1 \end{cases} \quad (9.7)$$

All of the exponentiations S^j and the divisions by j , $j = 1, 3, 5, \dots$, which make up for about two thirds of the total computational operations count, are very efficiently and accurately performed in LNS by simple shifts and additions. Therefore, the LNS polynomial implementation of the transcendental functions is a viable and exciting alternative to CORDIC vectoring. On the other hand, as Appendix A suggests, the hyperbolic trigonometric functions can be elegantly implemented in LNS, resulting in the same kind of savings. The architecture of a vectoring processor is abstracted in Figure 9.2 and the LNS procedure for polynomial evaluation of the angle Θ , where $S^2 < 1$ and the length of the power series terms is L , is given in Table 9.3. This computational procedure can be microcoded and invoked during run-time. A latency estimate is also computed, based on the assumption that the time $T_{A/S}$ required for addition or subtraction is



$$R = (x^2 + y^2)^{1/2}$$

θ = power expansion of equation (9.7)

FIGURE 9.2
LNS CORDIC Architecture

TABLE 9.3
LNS Arctan-Vectoring Procedure (see Figure 9.2)

Time	Operation	Control 1	P1	c_i	Control 2	P2
0	Subtract $y-x$	0	s	0	N/A	0
1	Iterate	N/A	$3s$	c_3	N/A	0
2	for	.	$3s$	c_3	1	s
5	angle	.	$3s$	c_3	-1	$P2+(3s+c_3)$
8	.	.	$5s$	c_5	1	$P2+(5s+c_5)$
.
.
.
$3L-6$.	N/A	$(2L+1)s$	c_{2L+1}	$(-1)^{L-2}$	$P2+(-1)^L[(2L-3)s+c_{2L-3}]$
$3L-3$.	1	$2x$	0	$(-1)^L$	$P2+(-1)^{L+1}[(2L-1)s+c_{2L-1}]$
$3L$.	2	$2y$.	1	$P2+(-1)^{L+3}[(2L+1)s+c_{2L+1}]$
$3L+1$	Output θ	N/A	N/A	.	1	$P2+(-1)^{L+3}[(2L+1)s+c_{2L+1}]$
$3L+3$		N/A	N/A	.	1	$2x$
$3L+6$		N/A	N/A	.	1	$2x+2y$
$3L+7$	Output R via S/R	N/A	N/A	.	1	$2x+2y$

three times the time $T_{M/D}$ required for multiplication or division. If the number of significant terms in (9.7) that are chosen to satisfy a prespecified precision metric is L , then the developed latency model yields a latency of $(3L + 6) T_{M/D}$ for the vectoring operation. The same operation performed using a 8088/8087 processor pair, would require about 17 multiply cycles to compute the angle and an additional 10.7 cycles to compute the radius; a total of 20.7 multiply cycles. It can be observed that the LNS approach is faster and more precise than the 8088/8087 pair. Experimental data found in Table 9.4 and produced using the code found in Appendix N, support the above analysis by showing that the relative error in the computation of the angle and the radius by polynomial expansion using LNS is significantly smaller than the one resulting from a FXP implementation of the CORDIC technique.

9.3 Pseudo Wigner-Ville Distribution

The Wigner-Ville Distribution (WVD) [Cla80, Fla84a, Vil48], and its evolved version (the Pseudo-WVD (PWVD)) are very good candidates when a time-frequency representation of a non-stationary signal is needed. Historically the Short Term Fourier Transform (STFT) [All77], has been the main analysis tool for studying signals with time-varying spectra. The STFT is premised on the so-called "constant frequency assumption." Therefore,

TABLE 9.4
Comparison of Errors in Transforming the Rectangular
Coordinates to Polar Ones, Using CORDIC and Through
Polynomial Expansion Using FXP and LNS Respectively

Relative Error $\times 10^{-6}$						Fractional Bits		
Angle θ			Radius R					
FXP	LNS	CORDIC	FXP	LNS	CORDIC	FXP	LNS	CORDIC
27560	876	350359	20550	314	285570	9	4	4
4369	705	19543	3932	75	6798	12	7	7
887	705	3266	556	64	618	12	10	10
835	705	1703	212	64	297	12	11	11
686	705	908	139	64	147	12	12	12

The Relative Error is computed as a noise-to-signal ratio, obtained by subtracting the results of the CORDIC technique and the FXP and LNS polynomial expansion values from the ones obtained using polynomial expansion with floating-point precision, and then dividing the differences with the latter.

the applied time-domain window must be sufficiently short over this interval, for the spectral signature to remain constant. A more general class of signals is the one of linear frequency (i.e., $\partial f(t)/\partial t = c_0 + c_1 t$). This class of signals can be efficiently processed using the PWVD and a Gaussian window (which will force the Wigner quasi probability density function to be nonnegative). In the discrete case, the Wigner distribution is given by

$$W(j, k) = 2 \sum_{n=-N/2}^{N/2} x(j, n) w(n) w_N^*(-n) w_N^{nk} \quad (9.8)$$

where $w(n)$ is a real window function, $w_N = \exp\{-2\pi j/N\}$ and $x(j, n)$ stands for the Wigner computational kernel $x(j+n) \cdot x^*(j-n)$. Willey et al. [Wil84] proposed a systolic architecture for the implementation of (9.8) for the unwindowed case. It can be observed that the PWVD is a complex multiply intensive statement. The Wigner kernel (windowed or not) is shown to be a massively complex multiplier intensive task. The data exhibited in Table 9.2 support the throughput advantage to be gained by LNS complex multipliers. To produce a Wigner kernel would require N complex multiplications, N real multiplications and N complex additions. Using the data from Table 9.2 it is found that the PWVD kernel can be created in the following times (in nanoseconds) for a conventional FLP system and LNS respectively

Conventional	$N(1262 + 320 + 222) = 1804 N$
LNS	$N(386 + 74 + 239) = 699 N$

In other words LNS is approximately 2.5 faster than conventional FLP processors. Of course the produced kernel can be processed by a standard N -point FFT. The latter has been studied, for the LNS case, by Swartzlander et al. [Swa83].

Recently, Flandrin et al. [Fla84b] published a variation on this theme by considering the signal to be analytic. More specifically, the real-valued signal $x(n)$ in (9.8) is mapped through a Hilbert transform into the series $z(n) = x(n) + jx_H(n)$, where x_H is the Hilbert-transformed signal defined (discrete case) by

$$x_H(n) = \sum_{m \neq n} x(m) \frac{\sin^2 \pi(m-n)/2}{\pi(m-n)/2} \quad (9.9)$$

If the temporal data extend by $\pm M$ delays about some sample t , and if a N -point harmonic spectrum is desired, then its Wigner signature is given by

$$W(t, k) = 4 \operatorname{Re} \left\{ \sum_{i=0}^{N-1} w^{ik} |h_N(i)|^2 \sum_{m=-M+1}^{M-1} g_M(m) z(t+m+i) z^*(t+m-i) \right\} - 2 |z(t)|^2 \quad (9.10)$$

where h_N and g_M are frequency and temporal windows respectively. The formula (9.10) allows for obtaining a $2N$ -point PWVD through a N -point FFT. A simplistic systolic WVD (not considering a window function) has been reported [Rab74]. Flandrin proposed a conventional architecture for implementing (9.10) using a TMS320 processor. A more elegant and faster systolic architecture, based on the developed LNS units is presented in Figure 9.3, where for clarity only the case $N=3$, $M=3$ is presented. Systolic primitives are utilized to implement the mapping $s \leftarrow s + ab$ required by the multiplier-intensive operations. This architecture is completely modular and general, in the sense that it accepts any desirable definition of window functions $h(i)$, $g(i)$. The implementation of the second window is particularly exciting. The architecture in Figure 9.3 is shown to be fast and fully utilizing the attributes of the LNS processor. The LNS approach offers as an side advantage the ability to alter the shape of certain window functions without any need for reprogramming. The preferred window function is the Gaussian window, mentioned above, and given by

$$G(i) = c e^{(-i^2/a^2)} \quad (9.11)$$

where c and a are design parameters. The LNS-Wigner processor would be presented with the value of

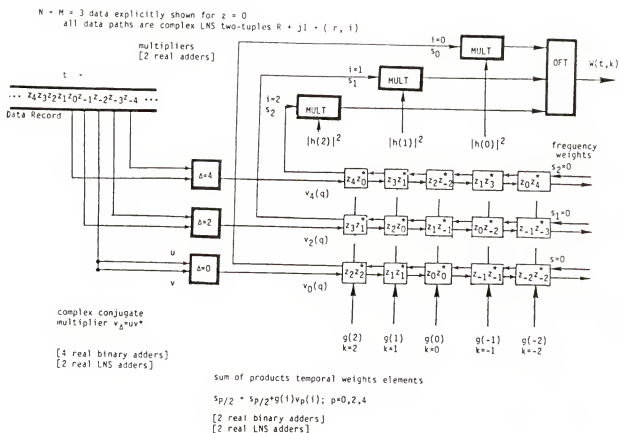


FIGURE 9.3
LNS Systolic Wigner Processor Architecture

$$g(i) = \text{lr}(G(i)) = \text{lr}(c) + m(i) ; m(i) = \frac{-i^2}{a^2} \text{lr}(e) \quad (9.12)$$

If a is reassigned a value $a' = da$; $d > 0$, then the new window weight is given by

$$g'(i) = g(i) + m(i) \left[\frac{1 - d^2}{d^2} \right] \quad (9.13)$$

If d is selected in such a way that $d = m(i) / [2^{\pm k} + m(i)]^k$ for any k , then the new window weight is obtained by a simple shift of the old weight by k bits.

9.4 Multiplicative FIR Filters

The linear Finite Impulse Response filters (FIR) play an important role in contemporary DSP. They are proven to be very effective in implementing linear phase digital filters. Linear phase behavior can be ensured by satisfying some straightforward coefficient symmetry conditions. However, compared to an Infinite Impulse Response (IIR) filter, an equivalent FIR will be of much greater order. Rabiner et al. [Rab74] have published empirical results which indicate that an FIR can be of an order several orders of magnitude bigger than the one of an equivalent IIR (having a similar ideal frequency response). In other words, the enhanced frequency-phase characteristics of the FIR filter are obtained through a degraded complexity-throughput performance. Low

complexity would require a few-multiplier (probably time-multiplexed) tap coefficient generation, while advanced throughput requires many concurrently operating multipliers.

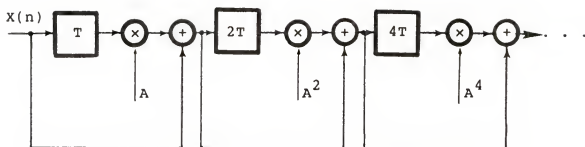
Recently, an alternative design methodology was proposed by Fam [Fam81] and extended by Taylor [Tay84]. Known as the Multiplicative FIR filter (MFIR), it is based on the use of the identity

$$\sum_{i=0}^{2^P-1} x^i = \prod_{i=0}^{P-1} (1 + x^{2^i}) \quad (9.14)$$

Replacing x by the quantity Az^{-1} (z is the usual z -transform variable), the following FIR transfer function results:

$$\sum_{i=0}^{2^P-1} (Az^{-1})^i = \prod_{i=0}^{P-1} \left(1 + (Az^{-1})^{2^i} \right) = \prod_{i=0}^{P-1} \omega_i(A, z) \quad (9.15)$$

where $\omega_i(A, z) = 1 + (Az^{-1})^{2^i}$. The impulse response of a system described by equation (9.15) is of exponential nature, given by the times series $\{x(i)\} = \{A^j\}$, for j assuming nonnegative integer values. The synthesis of a more general impulse response would require many concurrently operating MFIR stages. The potential advantage of (9.15) is that the operations count is much reduced. The conventional way requires 2^P coefficient scalings and 2^{P-1} additions, whereas the MFIR requires

CONVENTIONAL ARCHITECTURE

$$H_1(z) = 1 + Az^{-1} \quad H_2(z) = 1 + A^2z^{-2} \quad H_4(z) = 1 + A^4z^{-4}$$

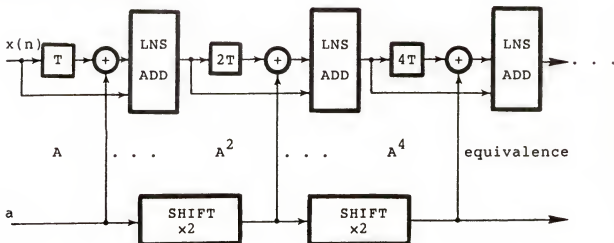
LNS ARCHITECTURE

FIGURE 9.4
LNS-MFIR Architecture

only P of those. An MFIR-LNS architecture is proposed in Figure 9.4, where $a = 1/r(A)$. It is well suited for this case, because LNS can also take advantage of the fact that the necessary forming of powers (i.e. $A^{\pm k}$; $k = 2^i$) can be done through elementary data shifts of the LNS exponent of A . Taylor [Tay84] has offered a design of an arbitrary transfer function by using a set of MFIRs connected in parallel and satisfying a l_2 optimization criterion. The architecture in Figure 9.4 is seen to possess a regular and modular form, suitable for VLSI design.

9.5 Echo Cancellation

Echo in telecommunications is undesirable and unavoidable [Gri84]. It is the result of impedance mismatch problems of the energy couplers between the four-wire circuit of the satellite communications link and the two-wire local circuit. Therefore, some energy on the satellite branch is coupled to the local branch and returned to the source as an echo. This echo (typically 11 dB down from the original signal) requires special treatment and it is usually done through an echo canceller [Wid75, Son67]. The suspiciously sounding, to the uninitiated, concept of echo cancellation--"To remove the echo subtract it," is based on mimicing the echo mapping function in the echo canceller device, which is connected in parallel to the echo path and synthesizes a replica of the echo, which is then subtracted from the combined echo

and near-end speech signal to obtain the near-end signal alone. The identification of the mapping function of the echo path is mostly done through an adaptive filter of one form or another and gradually matches the impulse response of the actual echo path. The principal algorithm in use today (mainly because of its simple structure and implementation) for echo cancellation is the Least-Mean-Squares or LMS algorithm [Wei79, Wid76]. The LMS echo canceller is shown in Figure 9.5, where $\{x(n)\}$ is the far-end signal source, $\{y(n)\}$ is the signal on the return path (which includes echo and the near-end signal $\{v(n)\}$), $\{\hat{y}(n)\}$ is its estimated version and $\{e(n)\}$ is the error signal that serves as feedback to the adaptive filter. The samples of the above waveforms at sampling instant kT are given by $x(k)$, $y(k)$ and $e(k)$ respectively. K is the loop gain or step size of the echo canceller, N is the number of taps in the echo canceller and $\hat{h}_i(k)$, $k = 1, \dots, N$ are the tap weights. The feedback signal of the echo canceller is given by

$$\begin{aligned}
 e(k) &= y(k) - \hat{y}(k) \\
 &= y(k) - \sum_{i=0}^{N-1} x(k-1)h(i,k) \\
 &= y(k) - \sum_{i=0}^{N-1} x(k-1)h(i,k) \qquad (9.16) \\
 &= y(k) - \sum_{i=0}^{N-1} x(k-1) \left[h(i,k-1) + 2Kx(k-1-i)e(k-1) \right]
 \end{aligned}$$

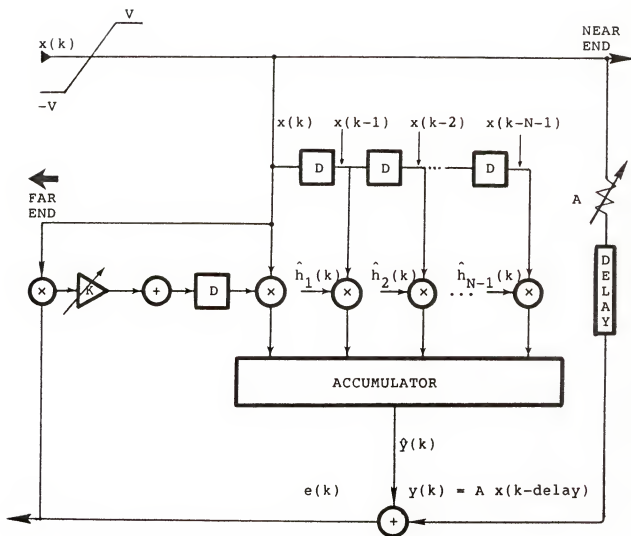


FIGURE 9.5
General Model of Echo Canceller

By adjusting the loop gain K , the problems of variation in far-end signal power and the double-talk can be addressed. It can be seen that this algorithm is very simple and requires only $O(2N)$ multiplications and $O(2N)$ additions per iteration for an N -tap filter. This makes it an ideal choice if the results of different implementation of a general algorithm via several arithmetic systems are to be considered in a balanced operation environment. Some experiments were performed along this line, through simulation, according to the source codes found in Appendix O. An experiment was held for each one of the floating-point, fixed-point and logarithmic echo canceller implementations. The far-end signal $x(k)$ was assumed to be Gaussian white noise with a mean of zero and limited between $\pm 2^p$, with p being defined interactively. The return-signal was supposed to be given by the product $Ax(k\text{-delay})$, where the constant A and the delay were again defined interactively. The criterion of comparison was the rate of convergence of the LMS algorithm, determined by the Mean Square Error (MSE). For an ensemble size of L , the MSE is given by

$$\text{MSE}(k) = \frac{\sum_{i=1}^L e_i^2(k)}{L} \quad (9.17)$$

where $e_i(k)$ is offered by equation (9.16). In Figure 9.6 the variable $\text{MSE}(k)$ versus time is plotted. The fractional

ECHO CANCELLER

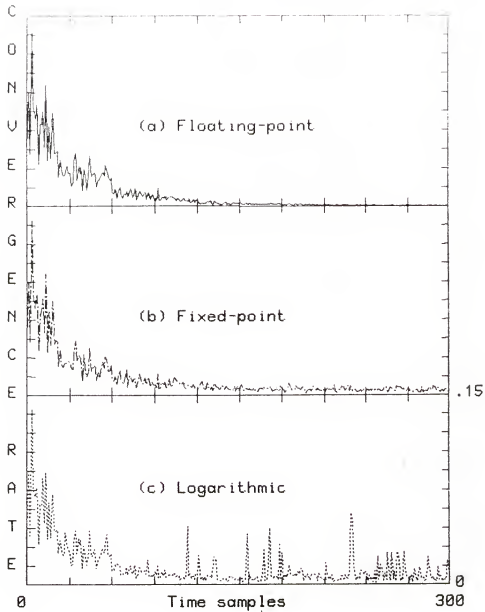


FIGURE 9.6

Convergence Curves for Three Echo Canceller Implementations:
 a) Floating-point, b) Fixed-point and c) LNS.
 Ensemble Size was 24

wordlengths used in the experiments were $16-p-1$ for the FXP canceller and $16-lr(p)-2$ for the LNS one, so that they would cover equivalent dynamic ranges. The values of the parameters involved were $K = 0.0125$, $A = 0.3$, the number of samples was 300, the number of taps $N = 50$, and the ensemble size $L = 24$. It can be observed that the 64-bit accurate FLP system (solid line) performs best of all (being slightly better), if it were assumed that all of the three systems require the same time for the execution of the operations involved. According to Table 9.2 though, this is not the case. In absolute time units the LNS echo canceller proves to converge at least 1.72 and 1.20 times faster than its FLP and FXP counterparts.

CHAPTER TEN IMPACT OF DESIGN ON ARCHITECTURE

Whereas the basic LNS unit is architected with internal (on-chip) memory, it is unrealistic to assume that a useful ARP-LNS device could be architected with multiple high-density large ROM arrays on the same chip. Most probably it will be architected with external memory, data paths and a central control, communication and computation chip, or C^5 unit. It will also contain a certain amount of fast cache memory. The C^5 architecture is shown in Figure 10.1, where the multiple internal paths support the following DSP operations supported by the indicated resources:

Operations	Resources
Multiply/divide	Adder
Add/subtract	Controller ¹ , Near-Zero PLA ² Comparator, Adder, ARP External memory
Square/Square root	Shift Register

¹ The ARP controller also checks for essential zeros.

² Near-zero Φ or Ψ mappings are "many to few."

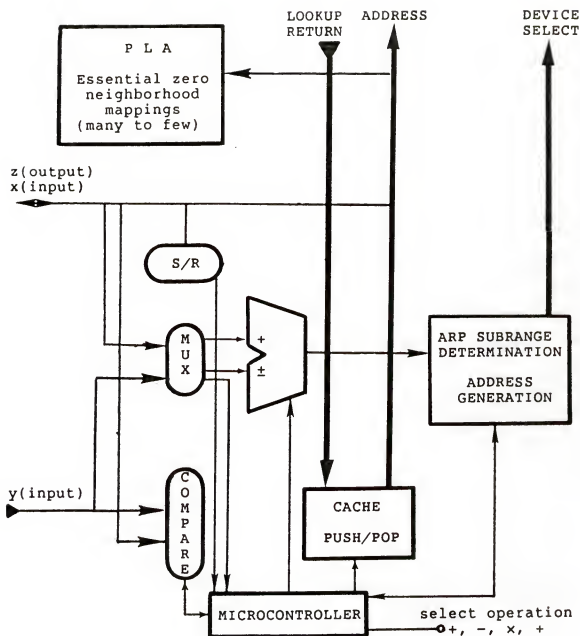


FIGURE 10.1
C⁵ Architecture

The external memory included in the ARP design presents some outstanding opportunities as well as some hazards when configured as two or three-dimensional memory architectures.

If permitted by the specific application and its data flow the C^5 architecture could allow for an LNS addition/subtraction scheme, which would be free from most memory dependencies. This can be done by first performing all the non additive operations, convert the results into a different number system (FLP, for example, which does not require any memory support) and use efficient operation-structuring methods, like trees, to perform fast additions. If it is required, the results could be again converted into LNS. Of course, this addition scheme can be justified only if the length of the addition sequence is large enough to write off the conversion overhead.

10.1 Two-Dimensional Memory Architectures

A C^5 -ARP would communicate, on demand, to one of a possibly large number of chips. This communication can be accomplished using a variety of possible networking schemes. At the high end of switching complexities are cross-bar networks. At the lower end are various networks [Fen81] like cubic rings, gamma networks, etc. Still, methods based on packet switching [Dia81] may be used. In a Dance Hall architecture, shown in Figure 10.2, switches are used to arbitrate memory tables to the ARP engines.

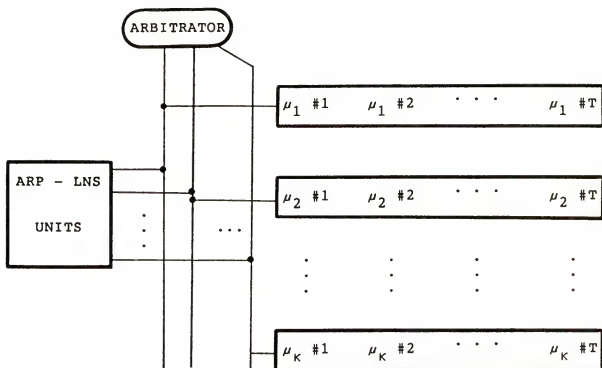


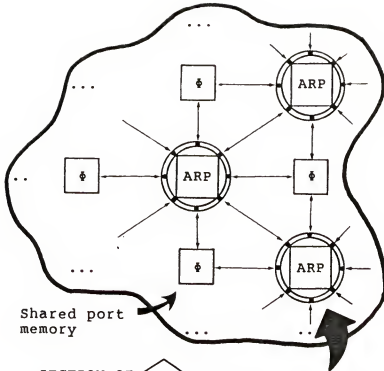
FIGURE 10.2
Dance Hall Architecture

The memory part consists of a total of κT chips organized along rows of identically programmed ROMS. Each row is responsible for LNS table look-ups from one of κ ARP intervals. Within each row there are T repetitions of the same table. During run-time, an ARP processor, if performing an addition requiring a table look-up support, will access one of the κT chips by issuing a select command. This command will have to specify a row (according to the ARP interval), enable an available chip within that row and finally present an address to the enabled chip. It is possible to define the number of memory rows and the number of identical tables within each row statistically, so that there is an optimum utilization of these tables. For example, for a three-partition case, an ARP would perform an addition by accessing one of the three memory rows, corresponding to the partition levels, according to the addition address. If the three tables are called μ_1 , μ_2 and μ_3 and if experimentation has determined that 60 percent of the calls are for μ_1 , 30 percent for μ_2 and 10 percent for μ_3 , then for a $\kappa T = 10$ chip design, the most reasonable assignment would be the one allocating 6 chips for the first row, 3 for the second and 1 for the third one, so that the number of memory calls would match statistically the number of available resources. It is also reasonable to assume (even without experimentation) that the number of tables required to be dedicated to the same address subrange will decrease as the subrange

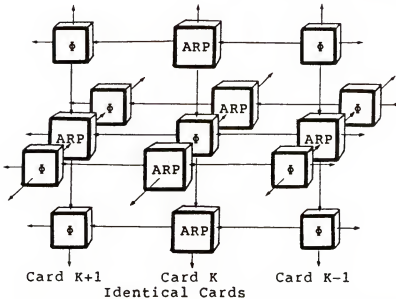
approaches the essential zero, especially if presorting of the addresses has already taken place. For optimal utilization, (as mentioned in the third step of the procedure for preparation of applications given in Chapter Nine) the total number κT of tables can be determined by using simulation to accumulate statistics regarding the application.

10.2 Three-Dimensional Memory Architectures

Because of the regularity of the ARP machine and the memory-data flow, the issue of integration can be taken to the next higher level. The previously developed memory architecture is basically planar (two-dimensional). What is now possible is to develop the 3-D system suggested in Figure 10.3. Besides communicating in a traditional plane (card level), multiple and regular paths are opened through plural intracard connectors in the third dimension (3-D space level). These paths have a bandwidth far in excess of that obtainable by using standard edge connectors. Using this architecture, effective nearest neighbor techniques can be implemented for ring and cube networks.



Circuit or Packet Switched Ring Topology



Cube equivalent of the circuit above.

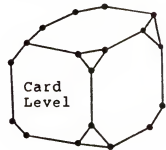


FIGURE 10.3
3-D C^5 ARP LNS Architecture

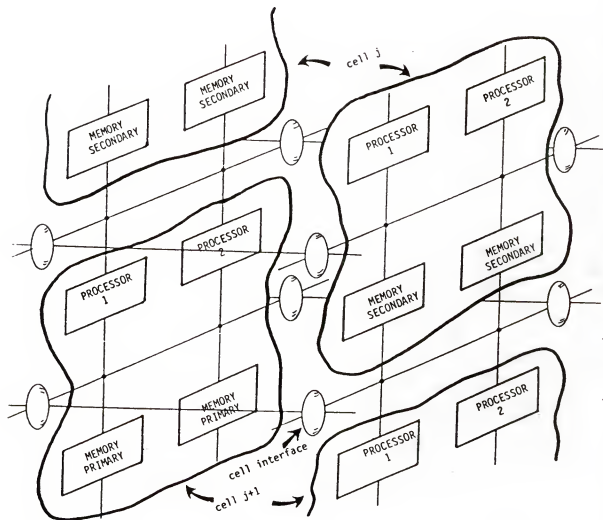
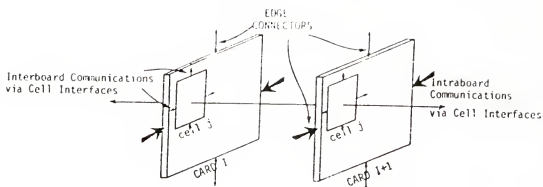


FIGURE 10.4
Intraboard Communications of 3-D Architectures
via Cell Interfaces.

10.3 Fault Tolerance

Globally, fault tolerance can be designed into the system by using a number of Error Correcting Codes (ECC) techniques and redundancy. However, an additional fault tolerance dimension can be achieved by attaching the C^5 units to a multidimensional memory array as the one shown in Figure 10.4. It can be seen there that a dual-port memory is associated with each processor, with one of its components being its primary choice for table look-ups, while the other one (belonging to an adjacent cell) is its secondary choice. The following fault-overcoming scheme can be employed. At the memory level, if an access to a memory table fails, data can be moved around the periphery of the memory table to a neighbor. If a dual-port memory (primary) fails, look-up operations can be assigned to its adjacent table (secondary) or passed through the complementing cell processor to its tables, or exported to an adjacent cell.

It is obviously desirable to be able to design a system, which can be rearchitected during run-time. The key to such a design is the ability to populate it with identical processor chips. Reconfiguration can be used for either rearchitecting a system for efficient execution of tasks or overlaying a degree of fault tolerance on a design. Instead of following the traditional notion of replacing fault components in-kind or allowing for

graceful degradation, one could try to explore another philosophy, borrowing elements from biology

"The ideal (or at least an excellent) reconfiguration system is the human body. If she or he loses a subsystem (e.g., sight) then the body will try to replace it in-kind (e.g., tissue regeneration) and lacking this ability, it will rewire the system to use the information of the functioning subsystems more efficiently (e.g., increased reliance on hearing)."

That is, instead of trying to replace the lost components with reallocated hardware, determine if this hardware reserve may be better used by increasing the power of a distinctly different subsystem. For example, if the memory data or address lines fail, the C^5 can still be of use by labeling it as supporting, say, only multiplication, division, squaring and other non-memory-intensive operations, instead of trying to force it back to a complete instruction set processor.

10.4 Memory Assignment Schemes

Several memory assignment schemes are candidates for employment. Among them are a stack pointer approach (on-chip) and a nearest neighbor one that would assign to the LNS processor the "closest" memory chip. The first approach (on-chip stack and microcontroller) allows for

considering a novel method for performing branching or servicing interrupts.

Suppose that a task is being processed until an event occurs, which requires the temporary termination of that task and the execution of the "new task" to commence. Classical wisdom would suggest that the states of the processor be saved for the old task, the new task be loaded and executed and finally the old task be restored and execution continued. However, based upon mathematical models (like the precedence relations) and the fact that the ARP design will probably provide for an on-chip microcontroller, it is possible to conceive a policy that would allow for transfer of control from the old to the new task and back again with a minimum of real time delay. This could prove to be a critical design objective in designing programmable signal processing systems, where the algorithm remains constant but the source of the input time series and the coefficient set can be defined by conditional branching tests or external control. In a multi-ARP system, one could start with the longest delay path to the output, complete that calculation while the next system set-up values are preloaded (by using probably doubly-buffered on-chip memory blocks) into the ARPs located at that level. Once the data exported from this longest delay path are assimilated to the next level, preload that stage with data from the interrupting task and continue. Using this method, it would appear as

though valid outputs could continuously flow from the processor. In other words this interrupt servicing technique introduces no latency.

Using the ARP, fast compact DSP systems can be designed, which will be able to operate over a large dynamic range on a low error budget. Many of these designs will comprise a number of identical and interconnected ARPs. These processor chips will have to share data. This sharing can take place in globally assigned shared memory, or through processor to processor communication. In general, for a design involving p processors (with p roughly in the range $2 \leq p \leq 32$), being able to fetch vectors of arbitrary length n from either local or shared global memory, Gannon et al. [Gan84] found that the execution time of dyadic operations is

$$r_{\infty}^{-1}(n + n_{\frac{1}{2}}^G)$$

if either operand is in global memory, while it is

$$r_{\infty}^{-1}(n + n_{\frac{1}{2}}^L)$$

if both operands are in local memory. Here r_{∞}^{-1} is the asymptotic performance rate for a one-processor design and $n_{\frac{1}{2}}$ is the vector length required to achieve half the asymptotic performance rate, assuming that local memory accesses have much less latency than global memory ones ($n_{\frac{1}{2}}^L \ll n_{\frac{1}{2}}^G$). It is obvious that the throughput can be increased by just including a sufficient number of registers in the design. Several other memory assignment schemes could also be considered as candidates.

CHAPTER ELEVEN CONCLUSIONS

The research conducted in the framework of this dissertation advanced the body of available knowledge about logarithmic number systems and arithmetic processors based on them.

Some new results are reported with regard to the execution of arithmetic operations in LNS. Emphasis was given to computation of trigonometric functions.

The choice of a specific base of logarithms for LNS was proven to be immaterial for a criterion based on the maximum allowable error for a given dynamic range and wordlength.

The problem of conversion to and from LNS was considered and a full-scale error analysis for the FLP to LNS conversion was provided. This analysis served as a basis for a subsequent stochastic analysis of the proposed advanced LNS processor designs, which are characterized by enhanced precision. Two such designs, the ARP and AMP, were examined. They were analyzed in terms of error budget, operational latency and amount of hardware involved. They were also compared to each other. The best of the two designs (ARP) proved to offer a precision enhancement equivalent of up to 50 percent

longer wordlength when compared to conventional contemporary LNS designs. Alone or integrated with other number systems (like the Signed-Digit Canonical system, whose effect on the ARP performance was studied in Chapter Nine) the ARP processor was used as a basis to form new processor designs, based on memory sharing techniques. Because of its modularity, simplicity of hardware and regularity of the data flow, it enabled the construction of 3-D systems, where multiple and regular paths are opened in the third dimension. Such designs facilitate communications by increasing the bandwidth and offer alternative solutions to the issues of fault tolerance and recovery.

The theoretical bound for the operations count of several DSP algorithms was targeted for the LNS processors and some innovative techniques were offered to approach it.

An applications development procedure was offered, which by way of summarizing the findings of this research, aspires to be optimal. Also, some LNS applications were examined and some exciting solutions were given to old problems (like the one of computing trigonometric functions).

A 20-bit LNS VLSI processor has been made feasible as a result of this research.

Still though, some questions remain open requiring their answer through future research. For example, the new

processor could form the spine of a highly parallel system architecture. While at an initial step, research should be directed towards the design of dedicated numeric intensive machines for such applications as inner-products and matrix transformations, it could be later expanded to a more general purpose setting. Since programs written for a Von Neumann machine are in general ill-conditioned for a parallel multi-processor engine, the software needs to be redesigned, in order to achieve high throughput and high processor utilization factors.

Of fundamental value will be to determine precisely the effect of adding cache memory to the ARP design, and optimally specify the size of this cache and the I/O requirements it imposes.

A determining factor to a successful performance of the LNS engine developed, for either the 2-D or 3-D case will be the presence of a predictive compiler, which will be capable of looking at the code (past, present and future) for intrinsic parallelism, the processor-memory (local or globally shared) availability, and the status of the network switches (busy, idle, faulty) imposed by communication link restrictions, and produce an automatically generated (AUTOGEN) code which will maximize the throughput. Part of this compiler tasks will be the optimization of the memory trafficking, the establishment of precedence relations and the minimization of interrupt latency.

Although some of the above research tasks can be undertaken through means of analytical mathematical models, several of those will be possible to carry out only through simulation. The end result would be to develop a VLSI processor floorplan, layout and simulation of an ARP system, and if applicable fabricate the design.

Using the VLSI LNS processors, a wide range of problems in general purpose scientific computing, DSP, robotics and so forth can be positively impacted. In particular, the developed architectures could significantly enhance real time adaptive filtering.

BIBLIOGRAPHY

- All77,
Allen, J. B., and L. R. Rabiner, "A Unified Approach to Short-time Fourier Analysis and Synthesis," Proceedings IEEE, vol. 65, no. 11, pp. 1558-1564, November 1977.
- Avi61,
Avizienis, A., "Signed-digit Number Representation for Fast Parallel Arithmetic," IRE Trans. Electronic Computers, vol. EC-10, no. 9, pp. 389-400, September 1961.
- Bec81,
Bechtolsheim, A., and T. Gross, "The Implementation of Logarithmic Arithmetic," Computer Systems Lab report, Stanford University, March 1981.
- Bem58,
Bemer, R. W., "A Subroutine Method for Calculating Algorithms," Communications ACM, vol. 1, pp. 5-7, May 1958.
- Bru75,
Brubaker, T. A., and J. C. Becker, "Multiplication Using Logarithms Implemented With Read-only Memory," IEEE Trans. Computers, vol. C-24, no. 8, pp. 761-765, August 1975.
- Can62,
Cantor, D., G. Estrin, and R. Turn, "Logarithmic and Exponential Function Evaluation in a Variable Structure Digital Computer," IRE Trans. Electronic Computers, vol. EC-11, no. 4, pp. 155-164, April 1962.
- Che72,
Chen, T. C., "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots," IBM J. Res. Development, vol. 16, pp. 380-388, July 1972.
- Cla80,
Claassen, T. A. C. M., and W. F. G. Mecklenbrauker, "The Wigner Distribution--A Tool for Time-frequency Signal Analysis Part I, Part II, Part III," Philips J. Res., vol. 35, pp. 217-250, 276-300, 372-383, 1980.
- Com65,
Combet, M., H. Van Zonneveld, and L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers," IEEE Trans. Elect. Computers, vol. EC-14, no. 6, pp. 863-867, December 1965.

- Dag59,
Daggett, D. H., "Decimal-binary Conversions in CORDIC,"
IRE Trans. Electronic Computers, vol. 8, no. 9, pp.
335-339, September 1959.
- Dia81,
Dias, D. M., and J. R. Jump, "Packet Switching Intercon-
nection Networks for Modular Systems," Computer, vol.
14, no. 12, pp. 43-53, December 1981.
- Duk71,
Duke, E. J., "RC Logarithmic Analog to Digital (LAD)
Conversion," IEEE Trans. Instrumentation and Measure-
ment, vol. IM-20, pp. 74-76, February 1971.
- Fam81,
Fam, A. F., "MFIR Filters: Properties and Applica-
tions," IEEE Trans., Acoust., Speech Signal Processing,
vol. ASSP-29, no. 12, December 1981.
- Fen81,
Feng, T. Y., "A Survey of Interconnection Networks,"
Computer, vol. 14, no. 12, pp. 12-27, December 1981.
- Fla84a,
Flandrin, P., and B. Escudie, "An Interpretation of the
Pseudo-Wigner-Ville Distribution," Signal Processing,
vol. 6, no. 1, pp. 27-36, January 1984.
- Fla84b,
Flandrin, P., W. Martin, and M. Zakharia, "On a
Hardware Implementation of the Wigner-Ville Transform,"
Int. Conf. on Digital Signal Processing, Florence, Ita-
ly, September 5-8, 1984.
- Fre85,
Frey, M. L., and F. J. Taylor, "A Table Reduction Tech-
nique for Logarithmically Architected Digital Filters,"
IEEE Trans. Acoust., Speech, Signal Processing, vol.
ASSP-33, no. 3, pp. 718-719, June 1985.
- Fur64,
Furet, J., B. Jacquemin, and J. Kaiser, "Les Techniques
Numeriques dans le Contrôle Nucleaire," Onde Elec-
tronique, vol. 44, pp. 758-763, July-August 1964.
- Gan84,
Gannon, D. B., and J. V. Rosendale, "On the Impact of
Communication Complexity on the Design of Parallel Nu-
merical Algorithms," IEEE Trans. Computers, vol. C-33,
no. 12, pp. 1180-1194, December 1984.

- Gri84,
Gritton, C. W. K., and D. W. Lin, "Echo Cancellation Algorithms," IEEE ASSP Magazine, vol. 1, no. 2, pp. 30-38, April 1984.
- Hal70,
Hall, E. L., D. D. Lynch, and S. J. Dwyer, III, "Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications," IEEE Trans. Computers, vol. C-19, no. 2, pp. 97-105, February 1970.
- Hav80,
Haviland, G. L., and A. A. Tuszynski, "A CORDIC Arithmetic Processor Chip," IEEE Trans. Computers, vol. C-29, no. 2, pp. 68-79, February 1980.
- Hwa79,
Hwang, K., Computer Arithmetic, John Wiley & Sons, New York, 1979.
- Kan73,
Kaneko, T., and B. Liu, "On Local Roundoff Errors in Floating-point Arithmetic," Journal ACM, vol. 20, pp. 391-398, July 1973.
- Kan85,
Kanopoulos, N., "A Bit-Serial Architecture for Digital Signal processing," IEEE Trans. Circuits and Systems, vol. CAS-32, no. 3, pp. 289-291, March 1985.
- Kar84,
Karp, A. H., "Exponential and Logarithm by Sequential Squaring," IEEE Trans. Computers, vol. C-33, no. 5, pp. 462-464, May 1984.
- Kin71,
Kingsbury, N. G., and P. J. W. Rayner, "Digital Filtering Using Logarithmic Arithmetic," Electronics Letters, vol. 7, pp. 56-58, Jan 28 1971.
- Kir77,
Kirkpatrick, D. G., and Z. V. Kedem, "Addition Requirements for Rational Functions," SIAM J. Comput., vol. 6, no. 1, pp. 188-199, 1977.
- Kur78,
Kurokawa, T., "Error Analysis of Digital Filters With Logarithmic Number System," Ph.D. Thesis, University of Oklahoma, Norman, 1978.

- Kur80,
Kurokawa, T., J. A. Payne, and S. C. Lee, "Error Analysis of Recursive Digital Filters Implemented With Logarithmic Number Systems," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-28, no. 6, pp. 706-715, December 1980.
- Lan85,
Lang, J. H., C. A. Zukowski, R. O. LaMaire, and C. H. An, "Integrated-circuit Logarithmic Arithmetic Unit," IEEE Trans. Computers, vol. C-34, no. 5, pp. 475-483, May 1985.
- Lee77a,
Lee, S. C., and A. D. Edgar, "Focus Microcomputer Number System," in Microcomputer design and applications, ed. S. C. Lee, pp. 1-40, Academic Press, New York, NY, 1977.
- Lee77b,
Lee, S. C., and A. D. Edgar, "The Focus Number System," IEEE Trans. Computers, vol. C-26, no. 11, pp. 1167-1170, November 1977.
- Lee79,
Lee, S. C., and A. D. Edgar, "Addendum to 'The Focus Number System'," IEEE Trans. Computers, vol. C-28, no. 9, p. 693, September 1979.
- Li80,
Li, P. P., "The Logarithm Arithmetic," display file # 3968, California Inst. of Tech., Computer Science Dept., Pasadena, CA, September 1980.
- Li81,
Li, P. P., "The Serial Log-machine," display file # 4517, California Inst. of Tech., Computer Science Dept., Pasadena, CA, May 1981.
- Liu69,
Liu, B., and T. Kaneko, "Error Analysis of Digital Filters Realized With Floating-point Arithmetic," Proceedings IEEE, vol. 57, no. 10, pp. 1735-1747, October 1969.
- Lo85,
Lo, H. Y., and Y. Aoki, "Generation of a Precise Binary Logarithm With Difference Grouping Programmable Logic Array," IEEE Trans. Computers, vol. C-34, no. 8, pp. 681-691, August 1985.

- Lug70,
Lugish, B. G., "A Class of Algorithms for Automatic Evaluation of Certain Elementary Function in a Binary Computer," Department of Computer Science Report, University of Illinois, Urbana, IL, June 1, 1970.
- Maj73,
Majithia, J. C., and D. Levan, "A Note on Base-2 Logarithm Computations," Proceedings IEEE, vol. 61, pp. 1519-1520, October 1973.
- Mar73,
Marasa, J. D., and D. W. Matula, "A Simulative Study of Correlated Error Propagation in Various Finite Precision Arithmetics," IEEE Trans. Computers, vol. C-22, no. 6, pp. 587-597, June 73.
- Mar72,
Marino, D., "New Algorithms for the Approximate Evaluation in Hardware of Binary Logarithms and Elementary Functions," IEEE Trans. Computers, vol. C-21, no. 12, pp. 1416-1421, December 1972.
- Meg62,
Meggitt, J. E., "Pseudo Division and Pseudo Multiplication Processes," IBM J. Res. Development, vol. 6, pp. 210-226, July 1962.
- Mit62,
Mitchell, J. N., "Computer Multiplication and Division Using Binary Logarithms," IRE Trans. Electronic Computers, vol. EC-11, pp. 512-517, August 1962.
- Mor73,
Morgenstern, J., "Note on a Lower Bound of the Linear Complexity of the Fast Fourier Transform," Journal ACM, vol. 20, no. 2, pp. 305-306, 1973.
- Mot55,
Motzkin, T. S., "Evaluation of Polynomials and Evaluation of Rational Values," Bull. Amer. Math. Soc., vol. 61, p. 163, 1955.
- Opp75,
Oppenheim, A. V., and R. W. Schafer, Digital Signal Processing, Prentice-Hall, Englewood Cliffs, N. J., 1975.
- Opp68,
Oppenheim, A. V., R. W. Schafer, and T. J. Stockham, "Nonlinear Filtering of Multiplied and Convolved Signals," Proceedings IEEE, vol. 56, pp. 1264-1291, August 1968.

- Ost54,
Ostrowski, A. M., "On Two Problems in Abstract Algebra Connected With Horner's Rule," in Studies presented to R. von Mises, pp. 40-48, Academic Press, New York, 1954.
- Pan83,
Pan, V. Y., "The Additive and Logical Complexities of Linear and Bilinear Arithmetic Algorithms," Journal of Algorithms, vol. 4, pp. 1-34.
- Pan66,
Pan, V. Y., "Methods of Computing Values of Polynomials," Russian Mathematical Surveys, vol. 21, pp. 105-136, 1966.
- Pap65,
Papoulis, A., Probability, Random Variables and Stochastic Processes, McGraw-Hill, 1965.
- Rab75,
Rabiner, L. R., and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall, Englewood Cliffs, N. J., 1975.
- Rab74,
Rabiner, L. R., J. F. Kaiser, O. Hermann, and M. T. Dolan, "Some Comparisons Between FIR and IIR Digital Filters," BSTJ, February 1974.
- Rei60,
Reitwiesner, G. W., "Binary Arithmetic," in Advances in Computers, vol. 1, pp. 261-265, Academic Press, New York, 1960.
- Sal54,
For a recent discussion see H. E. Salzer, "Radix Tables for Finding the Logarithm of any Number of 25 Decimal Places," in Tables of functions and of zeros of functions, pp. 143-144, National Bureau of Standards Applied Mathematics Series, New York, NY, 1954.
- San67,
Sandberg, I. W., "Floating-point Roundoff Accumulation in Digital Filter Realization," BSTJ, vol. 46, pp. 1775-1791, October 1967.
- Sar71,
Sarkar, B. P., and E. V. Krishnamurthy, "Economic Pseudo-division Processes for Obtaining Square Roots, Logarithm and Arctan," IEEE Trans. Computers, vol. C-20, pp. 1589-1593, December 1971.

- She83,
Shenoy, V. P., "Finite Register Effects in LMS Adaptive Digital Filters," Ph.D. Thesis, University of Cincinnati, 1983.
- She82,
Shenoy, V. P., and F. J. Taylor, "On Short Term Auto-correlation Implemented With Logarithmic Number System," Proc. 25th Midwest Symposium on Circuits and Systems, Michigan Tech., Houghton, MI, August 1982.
- She84,
Shenoy, V. P., and F. J. Taylor, "Error Analysis of LMS Adaptive Digital Filters implemented with logarithmic number system," Proc. 1984 IEEE Int. Conf. Acoust., Speech, Signal Processing, vol. 2, pp. 30.10.1-30.10.4, San Diego, CA, March 1984.
- Sic81a,
Sicuranza, G. L., "Two Dimensional Digital Filters With Finite Wordlength Coefficients on a Nonuniform Grid," European Conference on Circuit Theory and Design ECCTD '81, The Hague, August 25-28 1981.
- Sic81b,
Sicuranza, G. L., "2-D Digital Filters Using Logarithmic Number Systems," Electronics Letters, vol. 17, pp. 854-855, October 1981.
- Sic82,
Sicuranza, G. L., "On the Accuracy of 2-D Digital Filters Realizations Using Logarithmic Number Systems," Proc. 1982 IEEE Int. Conf. Acoust., Speech, Signal Processing, pp. 48-51, May 1982.
- Sic83,
Sicuranza, G. L., "On Efficient Implementations of 2-D Digital Filters Using Logarithmic Number Systems," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-31, no. 4, pp. 877-855, August 1983.
- Son67,
Sondhi, M. M., "An Adaptive Echo Canceller," BSTJ, vol. 46, no. 3, pp. 497-511, March 1967.
- Spe65,
Specker, W. H., "A Class of Algorithms for $\ln(x)$, $\exp(x)$, $\sin(x)$, $\cos(x)$, $\arctan(x)$ and $\operatorname{arccot}(x)$," IEEE Trans. Electronic Computers, vol. EC-14, pp. 85-86, 1965.

- Sri77,
Sripad, A. B., and D. L. Snyder, "A Necessary and Sufficient Condition for Quantization Errors to be Uniform and White," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-25, no. 5, pp. 442-448, October 1977.
- Sri78,
Sripad, A. B., and D. L. Snyder, "Quantization Errors in Floating-point Arithmetic," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-26, no. 5, pp. 456-463, October 1978.
- Swa79,
Swartzlander, E. E., "Comment on the Focus Number System," IEEE Trans. Computers, vol. C-28, no. 9, p. 693, September 1979.
- Swa80b,
Swartzlander, E. E., "Merged Arithmetic," IEEE Trans. Computers, vol. C-29, no. 10, October 1980.
- Swa75,
Swartzlander, E. E., and A. G. Alexopoulos, "The Sign/logarithm Number System," IEEE Trans. Computers, vol. C-24, no. 12, pp. 1238-1242, December 1975.
- Swa83,
Swartzlander, E. E., D. V. S. Chandra, H. T. Nagle, Jr., and S. A. Starks, "Sign/Logarithm Arithmetic for FFT Implementation," IEEE Trans. Computers, vol. C-32, no. 6, pp. 526-534, June 1983.
- Swa80a,
Swartzlander, E. E., and B. K. Gilbert, "Arithmetic for Ultra-High-Speed Tomography," IEEE Trans. Computers, vol. C-29, no. 5, pp. 341-353, May 1980.
- Tay83b,
Taylor, F. J., Digital Filter Design Handbook, Marcel Dekker Inc., New York, 1983.
- Tay83a,
Taylor, F. J., "An Extended Precision Logarithmic Number System," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-31, no. 1, pp. 232-234, February 1983.
- Tay84a,
Taylor, F. J., "A Distributed Arithmetic MFIR Filter," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-32, no. 1, pp. 186-189, February 1984.

- Tay84b,
Taylor, F. J., "A Logarithmic Arithmetic Unit for Signal Processing," Proc. 1984 IEEE Int. Conf. Acoust., Speech, Signal Processing, vol. 3, pp. 44.10.1-44.10.4, San Diego, CA, March 1984.
- Tay85,
Taylor, F. J., "A Hybrid Floating-point Logarithmic Number System Processor," IEEE Trans. Circuits and Systems, vol. CAS-32, no. 1, pp. 92-95, January 1985.
- Tod55,
Todd, J., "Motivations for Working in Numerical Analysis," Comm. Pure Appl. Math, vol. 8, pp. 97-116, 1955.
- Vil48,
Ville, J., "Theorie et Applications de la Notion de Signal Analytique," Cables et Transm., vol. 2A, no. 1, pp. 61-74, 1948.
- Vol59,
Volder, J. E., "The CORDIC Trigonometric Computing Technique," IRE Trans. Electronic Computers, vol. EC-8, pp. 330-334, September 1959.
- Wal71,
Walther, J. S., "A Unified Algorithm for Elementary Functions," Proceedings AFIPS 1971 Spring Joint Computer Conference, pp. 379-385, 1971.
- Wei79,
Weiss, A., and D. Mitra, "Digital Adaptive Filters: Conditions for Convergence, Rates of Convergence, Effects of Noise, and Errors Arising from the Implementations," IEEE Trans. Information Theory, vol. IT-25, no. 6, pp. 637-652, Nov79.
- Wid75,
Widrow, B., J. R. Glover, J. M. McCool, J. Kaunitz, C. Williams, R. H. Hearn, J. R. Zeidler, E. Dong, and R. C. Coodlin, "Adaptive Noise Cancelling: Principles and Applications," Proceedings IEEE, vol. 63, no. 12, pp. 1692-1716, December 1975.
- Wid76,
Widrow, B., J. M. McCool, M. L. Larimore, and C. R. Johnson, "Stationary and Non-stationary Characteristics of the LMS Adaptive Filter," Proceedings IEEE, vol. 64, no. 8, pp. 1151-1162, August 1976.

- Wil84,
Wiley, T., T. S. Durani, and R. Chapman, "An FFT Systolic Processor and its Applications," Proc. 1984 IEEE Int. Conf. Acoust., Speech, Signal Processing, San Diego, CA, March, 1984.
- Wil63,
Wilkinson, J. H., Rounding Errors in Algebraic Processes, Prentice-Hall, Englewood Cliffs, N. J., 1963.
- Win68,
Winograd, S., "A new Algorithm for Inner Product," IEEE Trans. Computers, vol. TC-17, pp. 693-694, 1968.
- Win70,
Winograd, S., "On the Number of Multiplications Necessary to Compute Certain Functions," Comm. Pure Appl. Math., vol. 23, pp. 165-179, 1970.

APPENDIX A
DERIVATION AND IMPLEMENTATION OF TRIGONOMETRIC
AND OTHER FUNCTIONS FOR LNS

Adopting the LNS representation for a real number $Z = S_z r^z$, introduced in Chapter Three, a variable Z can be defined as

$$Z = r^z = e^X = e^{r^X} \rightarrow Z = \frac{r^X}{\log(r)} = r^X \text{lr}(e)$$

A.1 Hyperbolic Trigonometric Functions

$$\text{For } H = r^h = \cosh(X) = \frac{e^X + e^{-X}}{2} = \frac{e^X(1 + e^{-2X})}{2} \rightarrow$$

$$\begin{aligned} h &= \text{lr}(e^X) + \text{lr}(1 + e^{-2X}) - \text{lr}(2) \\ &= z + \text{lr}(1 + r^{-2z}) - \text{lr}(2) \\ &= z + \Phi(2z) - \text{lr}(2) \end{aligned}$$

$$\text{For } M = r^\mu = \sinh(X) = \frac{e^X - e^{-X}}{2} = \frac{e^X(1 - e^{-2X})}{2} \rightarrow$$

$$\begin{aligned} \mu &= \text{lr}(e^X) + \text{lr}(1 - e^{-2X}) - \text{lr}(2) \\ &= z + \text{lr}(1 - r^{-2z}) - \text{lr}(2) \\ &= z + \Psi(2z) - \text{lr}(2) \end{aligned}$$

$$\text{For } T = r^\tau = \tanh(X) = \frac{\sinh(X)}{\cosh(X)} = \frac{M}{H} = \frac{r^\mu}{r^h} = r^{\mu-h} \rightarrow$$

$$\tau = \Psi(2z) - \Phi(2z)$$

$$\text{For } C = r^C = \coth(X) = \frac{\cosh(X)}{\sinh(X)} = \frac{H}{M} = \frac{r^h}{r^\mu} = r^{h-\mu} \quad \rightarrow$$

$$c = \Phi(2z) - \Psi(2z)$$

$$\text{For } A = r^a = \operatorname{sech}(X) = \frac{1}{\cosh(X)} = \frac{1}{H} = \frac{1}{r^h} = r^{-h} \quad \rightarrow$$

$$a = -z - \Phi(2z) + 1r(2)$$

$$\text{For } V = r^v = \operatorname{csch}(X) = \frac{1}{\sinh(X)} = \frac{1}{M} = \frac{1}{r^\mu} = r^{-\mu} \quad \rightarrow$$

$$v = -z - \Psi(2z) + 1r(2)$$

The architecture for hardware implementation of the above functions is already given in Figure 3.4. However, for clarity, the algorithms for $\cosh(X)$ and $\coth(X)$ are summarized below

$\cosh(X)$

- 1) Produce z
- 2) Shift z to the left by one bit
- 3) Present $2z$ to the table Φ and
- 4) Add $z + \Phi(2z) - 1r(2) \rightarrow h$

$\coth(X)$

- 1) Produce z
- 2) Shift z to the left by one bit
- 3) Present $2z$ to the tables Φ and Ψ
- 4) Subtract $\Phi(2z) - \Psi(2z) \rightarrow c$

The regularity of the data flow is obvious.

A.2 Other FunctionsMultiplication

$$\text{For } C = A \times B \quad \rightarrow \quad r^C \leftarrow r^a \times r^b \quad \rightarrow \quad c \leftarrow a + b$$

Division

$$\text{For } C = A \div B \quad \rightarrow \quad r^C \leftarrow \frac{r^a}{r^b} \quad \rightarrow \quad c \leftarrow a - b$$

Without loss of generality we can assume that $A \geq B$. Then

Addition

$$\begin{aligned} \text{For } E = A + B \quad \rightarrow \quad r^S &= r^a + r^b = r^a(1 + r^{b-a}) \quad \rightarrow \\ s &\leftarrow a + \Phi(b - a) \quad \text{with} \quad \Phi(b - a) = \text{lr}(1 + r^{b-a}) \end{aligned}$$

Subtraction

$$\begin{aligned} \text{For } \Delta = A - B \quad \rightarrow \quad r^\delta &= r^a - r^b = r^a(1 - r^{b-a}) \quad \rightarrow \\ \delta &\leftarrow a + \Psi(b - a) \quad \text{with} \quad \Psi(b - a) = \text{lr}(1 - r^{b-a}) \end{aligned}$$

For the special case when $A = B$, then $\delta \leftarrow 0$.

APPENDIX B
PROGRAM FOR SIMULATION OF THE FLP TO LNS ENCODER

```

/*****
*
* This program performs a simulation for the Error
* Analysis at the output of the Logarithmic Encoder.
*
* Total is the Sample size (30000 ?)
* seed1 & seed2 are random #s generator's initializers.
* The final result is the Histogram's value divided by
* (TOTAL * step) where step = (high - low) / HISTO_AXIS
* high and low define the dynamic range
* frac and frac1 define the fractional wordlength
* at the input and output of the log encoder table.
*
*****/

#include <stdio.h>
#include <math.h>
#define SIZE 40000
int HISTO_AXIS;
main()
{
    extern int HISTO_AXIS;
    int total;
    int i, low, high, seed1, seed2;
    double temp, low1, frac1, frac, high1;
    double rad, k, Quant1, A[SIZE], Quant;
    double alpha[SIZE], error[SIZE], y[SIZE], L[SIZE];
    double alphas[SIZE], error1[SIZE], y1[SIZE], L1[SIZE];
    char nam[20];
    short rand(), nfrom();
    printf("Enter HISTO_AXIS :0);
    scanf("%d", &HISTO_AXIS);
    printf("Enter rad, total Quant1 bits Quant2 bits:0);
    scanf("%f %d %f %f", &rad, &total, &frac, &frac1);
    printf("Enter seed1, seed2 :0);
    scanf("%d %d", &seed1, &seed2);
    printf("rad = %f total = %d and frac = %e0,
           rad, total, frac);
    Quant = pow(2., - frac);

```

```

Quant1 = pow(2., - frac1);
printf("Quant = %eQuant1 = %e0,
      Quant, Quant1);
(void) srand(seed1);
low = 5000;
high = 10000;
for (i = 1; i <= total; i++){
    alpha[i] = (double)(nfrom(low,high)) / (double) high;
    L[i] = log(alpha[i]) / log(rad);
}
(void) srand(seed2);
for (i = 1; i <= total; i++){
    alphas[i] = (double)(nfrom(low,high)) / (double) high;
    L1[i] = log(alphas[i]) / log(rad);
}
/* Histogram is performed below
   To check the uniformity of alpha */
low1 = .5;
high1 = 1.;
histo(low1, high1, total, "alpha", alpha);

/* Histogram is performed below
   to check the uniformity of alpha */
low1 = log(.5) / log(rad);
high1 = 0.;
histo(low1, high1, total, "L", L);

for (i = 1; i <= total; i++){
    alpha[i] = floor(0.5 + alpha[i] / Quant) * Quant;
    y[i] = log(alpha[i]) / log(rad);
}
for (i = 1; i <= total; i++){
    alphas[i] = floor(0.5 + alphas[i] / Quant) * Quant;
    y1[i] = log(alphas[i]) / log(rad);
}

/* Histogram is performed below
   for alpha */
low1 = 0.5;
high1 = 1.;
histo(low1, high1, total, "alpha *", alpha);

/* Histogram is performed below
   for y=lr(a) */
low1 = log(low1) / log(rad);
high1 = log(high1) / log(rad);
histo(low1, high1, total, "y", y);

for (i = 1; i <= total; i++){
    y[i] = floor(0.5 + y[i] / Quant1) * Quant1;
    error[i] = y[i] - L[i];
}

```

```

/* Histogram is performed below
   for the discrete values of y */
lowl = lowl - Quantl / 2.;
highl = highl + Quantl / 2.;
histo(lowl, highl, total, "y ", y);

/* Histogram is performed below
   to define the p.d.f. of the error */
lowl = log(1. - Quant) / log(rad) - Quantl / 2.;
highl = log(1. + Quant) / log(rad) + Quantl / 2.;
histo(lowl, highl, total, "error", error);
}

/* The function nfrom returns a random number
   between low and high inclusive */
short nfrom(low, high)
register short low, high;
{
    short rand();
    register short nb = high - low + 1;
    return(rand() % nb + low);
}

/* Histogram function
   In the calling function the following should be
   specified : a) HISTO_AXIS : < 104 */
histo(low, high, total, name, array)
#define CHARSIZE 20
int total;
double low, high;
char name[CHARSIZE];
double array[SIZE];
{
    extern int HISTO_AXIS;
    int i, j, count[110];
    double step, fit, m;
    char star, star1;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outlogrm", "w");
    star = ' ' ;
    star1 = 'o' ;
    printf(" Histogram for %s 0, name);
    for (j = 0; j <= HISTO_AXIS + 5; j++){
        /* 5 stands for a nicer table */
        count[j] = 0;
    }
    step = (high - low) / HISTO_AXIS;
    printf("step = %f0, step);
    for (j = 1; j <= total; j++){
        if (array[j] < low - (abs(low) / 1.e+06))
            printf(" Low FIT for histogram %s = %f0,
                    name, array[j]);
        else if (array[j] > high)

```

```

        printf(" High fit for histogram %s = %f0,
               name,array[j]);
    else
        for (i = 1; i <= HISTO_AXIS; i++){
            if ((array[j] < (Low + i * step)) &&
                (array[j] >= (low + (i - 1) * step)))
                count[i] = count[i] + 1;
        }
    }
    fit = 0.;
    for (i = 1; i <= HISTO_AXIS; i++){
        fprintf(fp,"%d%e0, I, count[i] / (total * step));
        if (fit < count[i])
            fit = count[i];
    }
    fclose(fp);
    printf(" Low = %e total= %d High = %e fit=%f0,
           low, total, high, fit);
    for (i = 1; i <= HISTO_AXIS; i = i + 5)
        printf("count[%d] =%d%d%d%d%0,
               i, count[i],
               count[i+1], count[i+2], count[i+3],count[i+4]);
    for (i = 1; i <= HISTO_AXIS; i++){
        printf("%d", i);
        for (m = 1; m <= 75. * (count[i] / fit); m++)
            printf("%c",star);
        printf("%c0,star1);
    }
}

```

APPENDIX C
CODE FOR GENERATION OF THEORETICAL CURVE OF THE
p.d.f. OF THE LOG ENCODER ERROR. CASE i)

```

/*****
*
*   This program computes the theoretical p.d.f.
*   for the ERROR E at the output of the
*   FLP to LNS Encoder.
*   It passes its output to the file 'outfE'
*   to be used for plotting purposes.
*   Accepts the base (radix), Fractional wordlength (frac)
*   and the Total number of points to assume and computes
*   the values for the p.d.f. of E
*   It assumes that there is one bit more at the output
*   of the mapping memory table.
*
*****/

```

```

#include <stdio.h>
#include <math.h>
#define arraysize 1000
main()
{
    int j, k;
    double i, frac, high, low, step, power, a[arraysize];
    double temp, rad, Q, total, Q1, lnr;
    char star, star1;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outfE", "w");
    star = ' ';
    star1 = 'o';
    printf("Enter radix, frac, total :0);
    scanf("%f %f %f", &rad, &frac, &total);
    Q = pow(2., - frac);
    Q1 = pow(2., - (frac + 1.));
    lnr = log(rad);
    low = log(1. - Q) / lnr - Q1 / 2.;
    high = log(1. + Q) / lnr + Q1 / 2.;
    printf(" low = %f high = %f0, low, high);
    step = (high - low) / total;
    j = 1;

```

```

for (i = low; i <= (high + step); i += step){
    if ((i > log(1. - Q) / lnr - Q1 / 2.) &&
        (i <= log(1. - Q) / lnr + Q1 / 2.)){
        a[j] = (Q / (4. * Q1));
        a[j] *= (1. / (1. - pow(rad, i + Q1 / 2.))
            - 1. / Q);
        a[j] -= (1. / (4. * Q * Q1)) *
            (pow(rad, i + Q1 / 2.) + Q - 1.);
    }
    else if ((i > log(1. - Q) / lnr + Q1 / 2.)
        && (i <= log(1. - Q / 2.) / lnr - Q1 / 2.)){
        a[j] = Q / (4. * Q1);
        a[j] *= 1. / (1. - pow(rad, i + Q1 / 2.))
            - 1. / (1. - pow(rad, i - Q1 / 2.));
        a[j] -= (1. / (4. * Q * Q1)) *
            (pow(rad, i + Q1 / 2.) - pow(rad, i - Q1 / 2.));
    }
    else if ((i > log(1. - Q / 2.) / lnr - Q1 / 2.)
        && (i <= log(1. - Q / 2.) / lnr + Q1 / 2.)){
        a[j] = (Q / (4. * Q1)) * (2. / Q - 1. / (1. - pow(
            rad, i - Q1 / 2.))) - (1. / (4. * Q * Q1))
            * (1. - Q / 2. - pow(rad, i - Q1 / 2.));
        a[j] += (3. / (4. * Q * Q1)) *
            (pow(rad, i + Q1 / 2.) - 1. + Q / 2.);
    }
    else if ((i > log(1. + Q / 2.) / lnr + Q1 / 2.)
        && (i <= log(1. + Q / 2.) / lnr - Q1 / 2.)){
        a[j] = (3. / (4. * Q * Q1)) * (pow(
            rad, i + Q1 / 2.) - pow(rad, i - Q1 / 2.));
    }
    else if ((i > log(1. + Q / 2.) / lnr - Q1 / 2.)
        && (i <= log(1. + Q / 2.) / lnr + Q1 / 2.)){
        a[j] = (3. / (4. * Q * Q1)) * (- pow(
            rad, i - Q1 / 2.) + 1. + Q / 2.);
        a[j] += (Q / (4. * Q1)) * (2. / Q + 1. / (1.
            - pow(rad, i + Q1 / 2.))) - (1. / (4. * Q * Q1))
            * (- 1. - Q / 2. + pow(rad, i + Q1 / 2.));
    }
    else if ((i > log(1. - Q / 2.) / lnr + Q1 / 2.)
        && (i <= log(1. + Q) / lnr - Q1 / 2.)){
        a[j] = (Q / (4. * Q1)) * (1. / (1. - pow(rad,
            i + Q1 / 2.)) - 1. / (1. - pow(rad, i - Q1 /
            2.))) - (1. / (4. * Q * Q1)) * (pow(rad, i
            + Q1 / 2.) - pow(rad, i - Q1 / 2.));
    }
    else if ((i > log(1. + Q) / lnr - Q1 / 2.)
        && (i <= log(1. + Q) / lnr + Q1 / 2.)){
        a[j] = (Q / (4. * Q1)) * (- 1. / Q - 1. / (1.
            - pow(rad, i - Q1 / 2.))) - (1. / (4. * Q * Q1))
            * (1. + Q - pow(rad, i - Q1 / 2.));
    }
    printf(" a[%d] at %f = %e0, j, i, a[j]);
    fprintf(fp, "%d%e0, j, a[j]);

```

```
        j++;
    }
    fclose(fp);
    temp = 0;
    for (j = 1; j <= total; j = j++)
        if (a[j] > temp) temp = a[j];
    if (temp >= 75.)
        for (j = 1; j <= total; j = j++)
            a[j] = 75. * (a[j] / temp);
    for (j = 1; j <= total; j = j++){
        printf("%d", j);
        for (k = 1; k <= a[j]; k++)
            printf("%c", star);
        printf("%c0", star1);
    }
}
```

APPENDIX D
CODE FOR GENERATION OF THEORETICAL CURVE OF THE
p.d.f. OF THE LOG ENCODER ERROR. CASE ii)

```

/*****
*
* This program computes the theoretical p.d.f.
* for the ERROR E at the output of the
* FLP to LNS Encoder.
* Accepts the base (radix), Fractional wordlength (frac)
* and the Total number of points to assume and computes
* the values for the p.d.f. of E
* It assumes that there is an equal number of bits
* available at the input and output
* of the mapping memory table.
* It passes its output to the file 'outfE='
* to be used for plotting purposes.
*
*****/

```

```

#include <stdio.h>
#include <math.h>
#define arraysize 1000
double lg(), Q, Q1, int_A(), int_B(), rad;
main()
{
    int j, k;
    double i, frac, high, low, step, power, a[arraysize];
    double temp, total;
    char star, star1;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outfE=", "w");
    star = ' ';
    star1 = 'o';
    printf("Enter radix, frac, total :0);
    scanf("%f %f %f", &rad, &frac, &total);
    Q = pow(2., - frac);
    Q1 = Q;
    low = lg(1. - Q) - Q1 / 2.;
    high = lg(1. + Q) + Q1 / 2.;
    printf(" low = %f high = %f0, low, high);
    step = (high - low) / total;

```



```

j = 1;
for (i = low; i <= (high + step); i += step){
    if ((i > lg(1. - Q) - Q1 / 2.) &&
        i <= lg(1. - Q / 2.) - Q1 / 2.)){
        a[j] = (1. / Q1) * (int_A(i + Q1 / 2., lg(1. - Q)));
    }
    else if ((i > lg(1. - Q / 2.) - Q1 / 2.)
        && (i <= lg(1. - Q) + Q1 / 2.)){
        a[j] = (1. / Q1) * (int_A(lg(1. - Q / 2.), lg
            (1. - Q)) + int_B(i + Q1 / 2., lg(1 - Q / 2.)));
    }
    else if ((i > lg(1. - Q) + Q1 / 2.)
        && (i <= lg(1. - Q / 2.) + Q1 / 2.)){
        a[j] = (1. / Q1) * (int_A(lg(1. - Q / 2.), i -
            Q1 / 2.) + int_B(i + Q1 / 2., lg(1. - Q / 2.)));
    }
    else if ((i > lg(1. - Q / 2.) + Q1 / 2.)
        && (i <= lg(1. + Q / 2.) - Q1 / 2.)){
        a[j] = (1. / Q1) *
            int_B(i + Q1 / 2., i - Q1 / 2.);
    }
    else if ((i > lg(1. + Q / 2.) - Q1 / 2.)
        && (i <= lg(1. + Q) - Q1 / 2.)){
        a[j] = (1. / Q1) * (int_B(lg(1. + Q / 2.), i -
            Q1 / 2.) + int_A(i + Q1 / 2., lg(1. + Q / 2.)));
    }
    else if ((i > lg(1. + Q / 2.) + Q1 / 2.)
        && (i <= lg(1. + Q) + Q1 / 2.)){
        a[j] = (1. / Q1) * int_A(lg(1. + Q), i - Q1 / 2.);
    }
    printf(" a[%d] at %f = %e0, j, i, a[j]);
    fprintf(fp, "%d%e0, j, a[j]);
    j++;
}
fclose(fp);
temp = 0;

for (j = 1; j <= total; j = j++)
    if (a[j] > temp)
        temp = a[j];

if (temp >= 75.)
for (j = 1; j <= total; j = j++)
    a[j] = 75. * (a[j] / temp);

for (j = 1; j <= total; j = j++){
    printf("%d", j);

```

```

        for (k = 1; k <= a[j]; k++)
            printf("%c",star);

        printf("%c0,star1);
    }
}
double lg(arg)
double arg;
{
    return(log(arg) / log(rad));
}
double int_A(up, down)
double up, down;
{
    double temp;
    if (up > down) {
        temp = (Q / 4.) * (1. / (1. - pow(rad, up)) -
            1. / (1. - pow(rad, down)));
        temp -= (1. / (4. * Q)) * (pow(rad, up)
            - pow(rad, down));
    }
    else
        temp = 0.;
    return(temp);
}
double int_B(up, down)
double up, down;
{
    double temp;
    if (up > down) {
        temp = (3. / (4. * Q)) * (pow(rad, up)
            - pow(rad, down));
    }
    else
        temp = 0.;
    return(temp);
}

```

APPENDIX E
CODE FOR GENERATION OF THEORETICAL CURVE OF THE
p.d.f. OF THE LOG ENCODER ERROR. CASE iii)

```

/*****
*
* This program computes the theoretical p.d.f.
* for the ERROR E at the output of the
* FLP to LNS Encoder.
* Accepts the base (radix), Fractional wordlength (frac)
* and the Total number of points to assume and computes
* the values for the p.d.f. of E for the second case
* (when  $Q_0 = Q_1 + 1$ )
* It assumes that there is an equal number of bits
* available at the input and output
* of the mapping memory table.
* It passes its output to the file 'outfE_sec'
* to be used for plotting purposes.
*
*****/

```

```

#include <stdio.h>
#include <math.h>
#define arraysize 1000
double lg(), Q, Q1, int_A(), int_B(), rad;
main()
{
    int j, k;
    double i, frac, high, low, step, power, a[arraysize];
    double temp, total;
    char star, star1;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outfE_sec", "w");
    star = ' ';
    star1 = 'o';
    printf("Enter radix, frac, total :0);
    scanf("%f %f %f", &rad, &frac, &total);
    Q = pow(2., - frac);
    Q1 = pow(2., - frac + 1.);
    low = lg(1. - Q) - Q1 / 2.;
    high = lg(1. + Q) + Q1 / 2.;
    step = (high - low) / total;

```

```

j = 1;
for (i = low; i <= (high + step); i += step){
    if ((i > lg(1. - Q) - Q1 / 2.) &&
        (i <= lg(1. - Q / 2.) - Q1 / 2.)){
        a[j] = (1. / Q1) *
            (int_A(i + Q1 / 2., lg(1. - Q)));
    }
    else if ((i > lg(1. - Q / 2.) - Q1 / 2.)
        && (i <= lg(1. - Q) + Q1 / 2.)){
        a[j] = (1. / Q1) * (int_A(lg(1. - Q / 2.), lg
            (1. - Q)) + int_B(i + Q1 / 2., lg(1 - Q / 2.)));
    }
    else if ((i > lg(1. - Q) + Q1 / 2.)
        && (i <= lg(1. + Q / 2.) - Q1 / 2.)){
        a[j] = (1. / Q1) * (int_A(lg(1. - Q / 2.), i -
            Q1 / 2.) + int_B(i + Q1 / 2., lg(1. - Q / 2.)));
    }
    else if ((i > lg(1. + Q / 2.) - Q1 / 2.)
        && (i <= lg(1. - Q / 2.) + Q1 / 2.)){
        a[j] = (1. / Q1) * (int_A(lg(1. - Q / 2.), i -
            Q1 / 2.) + int_B(lg(1. + Q / 2.), lg(1. - Q /
            2.)) + int_A(i + Q1 / 2., lg(1. + Q / 2.)));
    }
    else if ((i > lg(1. - Q / 2.) + Q1 / 2.)
        && (i <= lg(1. + Q) - Q1 / 2.)){
        a[j] = (1. / Q1) * (int_B(lg(1. + Q / 2.), i -
            Q1 / 2.) + int_A(i + Q1 / 2., lg(1. + Q / 2.)));
    }
    else if ((i > lg(1. + Q) - Q1 / 2.)
        && (i <= lg(1. + Q / 2.) + Q1 / 2.)){
        a[j] = (1. / Q1) *
            (int_B(lg(1. + Q / 2.), i -
            Q1 / 2.) + int_A(lg(1. + Q), lg(1. + Q / 2.)));
    }
    else if ((i > lg(1. + Q / 2.) + Q1 / 2.)
        && (i <= lg(1. + Q) + Q1 / 2.)){
        a[j] = (1. / Q1) *
            int_A(lg(1. + Q), i - Q1 / 2.);
    }
    printf(" a[%d] at %f = %e0, j, i, a[j]);
    fprintf(fp, "%d%e0, j, a[j]);
    j++;
}

fclose(fp);
temp = 0;

for (j = 1; j <= total; j = j++)
    if (a[j] > temp)
        temp = a[j];

if (temp >= 75.)
    for (j = 1; j <= total; j = j++)
        a[j] = 75. * (a[j] / temp);

```

```

    for (j = 1; j <= total; j = j++){
        printf("%d", j);

        for (k = 1; k <= a[j]; k++)
            printf("%c",star);

        printf("%c0",star1);
    }
}

double lg(arg)
double arg;
{
    return(log(arg) / log(rad));
}

double int_A(up, down)
double up, _down;
{
    double temp;
    if (up > down) {
        temp = (Q / 4.) * (1. / (1. - pow(rad, up)) -
                     1. / (1. - pow(rad, down)));
        temp -= (1. / (4. * Q)) * (pow(rad, up)
                                   - pow(rad, down));
    }
    else
        temp = 0.;
    return(temp);
}

double int_B(up, down)
double up, _down;
{
    double temp;
    if (up > down)
        temp = (3. / (4. * Q)) * (pow(rad, up)
                                   - pow(rad, down));
    else
        temp = 0.;
    return(temp);
}

```

APPENDIX F
CODE FOR GENERATION OF APPROXIMATION CURVE OF THE
p.d.f. OF THE LOG ENCODER ERROR

```

/*****
*
*   This program computes the Theoretical
*   Approximation of the theoretical p.d.f. for the
*   ERROR E at the output of the FLP to LNS Encoder.
*
*   Accepts the base (radix), Fractional wordlength (frac)
*   and the Total number of points to assume and computes
*   the values for the p.d.f. of E for the second case
*   (when  $Q_0 = Q_i + 1$ ).
*
*   It assumes that there is an equal number of bits
*   available at the input and output
*   of the mapping memory table.
*   It passes its output to the file 'outappr'
*   to be used for plotting purposes.
*
*****/

```

```

#include <stdio.h>
#include <math.h>
#define arraysize 1000
main()
{
    int j, k;
    double i, rad, frac, high, low, power, a[arraysize];
    double step, temp, Q, Lamda, Kappa, x1K;
    double x2K, x1L, x2L, total, Q1, lnr;
    char star, star1;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outappr", "w");
    star = ' ';
    star1 = 'o';
    printf("Enter radix, frac, total :0);
    scanf("%f %f %f", &rad, &frac, &total);
    Q = pow(2., - frac);
    Q1 = pow(2., - (frac + 1.));
    lnr = log(rad);

```

```

low = log(1. - Q) / lnr - Q1 / 2.;
high = log(1. + Q) / lnr + Q1 / 2.;
printf(" low = %f high = %f0, low, high);
Kappa = (3. * (1. - pow(rad, - Q1)) * (1. + Q / 2.)) /
(4. * Q * Q1 * pow(rad, Q1 / 2.));
Lamda = (3. * (1. - Q / 2.) *
(pow(rad, Q1) - 1.)) / (4. * Q * Q1);
x1L = low;
x2L = log(1. - Q / 2.) / lnr + Q1 / 2.;
x2K = log(1. + Q / 2.) / lnr - Q1 / 2.;
x1K = high;
step = (high - low) / total;
j = 1;
for (i = low; i <= (high + step); i += step){
    if ((i > low) &&
        (i <= log(1. - Q / 2.) / lnr + Q1 / 2.)){
        a[j] = (i - x1L) * Lamda / (x2L - x1L);
    }
    else if ((i > log(1. - Q / 2.) / lnr + Q1 / 2.)
        && (i <= log(1. + Q / 2.) / lnr - Q1 / 2.)){
        a[j] = (3. / (4. * Q * Q1)) *
            (pow(rad, i + Q1 / 2.) - pow(rad, i - Q1 / 2.));
    }
    else if ((i > log(1. + Q / 2.) / lnr - Q1 / 2.)
        && (i <= high)){
        a[j] = (i - x1K) * Kappa / (x2K - x1K);
    }
    fprintf(fp, "%d%e0, j, a[j]);
    j++;
}
fclose(fp);
temp = 0;
for (j = 1; j <= total; j = j++){
    if (a[j] > temp)
        temp = a[j];
}
if (temp >= 75.)
for (j = 1; j <= total; j = j++)
    a[j] = 75. * (a[j] / temp);
for (j = 1; j <= total; j = j++){
    printf("%d", j);
    for (k = 1; k <= a[j]; k++){
        printf("%c", star);
    }
    printf("%c0, star1);
}
}
}

```

APPENDIX G
CODE FOR GENERATION OF THE ESSENTIAL ZEROS ENTRIES
OF TABLE 5.1

```

/*****
*
*      This program computes the Essential Zeros
*      for the addition and subtraction mappings
*      if the base and the number of fractional
*      bits of the LNS are known.
*
*****/

```

```

#include <stdio.h>
#include <math.h>
double M, radix;
main()
{
    double v();
    int c;
    extern double M, radix;

    while ((c = getchar()) != EOF) {
        printf("enter radix r : , M :0);
        scanf("%f %f", &radix, &M);
        printf("zero = %f0,v());
        c = getchar();
    }
}

double v()      /* Computes the essential zero for
                  given # of fractional bits M,
                  and given radix r.          */
{
    extern double M, radix;
    double V;
    V = - log(pow(radix,pow(2.,- M)) - 1.) / log(radix);
    return (V);
}

```


APPENDIX H
CODE FOR GENERATION OF THEORETICAL CURVE OF THE
ERROR p.d.f. OF THE LOGARITHMIC ADDER

```

/*****
*
* This program computes the theoretical p.d.f.
* for the ERROR E at the output of the
* Logarithmic Adder for a specific address v.
* Accepts the base (radix), Fractional wordlength (frac)
* at the output of the memory table
* Iex is the number of integer (only) bits that the LNS
* exponent x is using.
* v determines the table address
* It passes its output to the file 'outdensity'
* to be used for plotting purposes.
*
*****/

```

```

#include <math.h>
#include <stdio.h>
#define SIZE 1000
double radix, lnr, Q4, QL;
main()
{
    extern double radix, lnr, Q4, QL;
    double temp, density[SIZE], a1, L, a2, b1, b2, c1, c2;
    double EL(), EL1(), EL2(), efl(), ef2(), ef3();
    char star, star1;
    int eps, k;
    double Iex, suma, step, high, sumb, frac;
    int condition;
    double sum1, sum2, dd1, dd2, v, sum3, delta1, delta2;
    double i, sumc, j, Prob1, Prob2, Prob3, dd3, l, m;
    double lmean(), total, low, Zero, Ivar(), Prob(), sum;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outdensity", "w");

    printf(" frac, radix, v, Iex ,L, total0);
    scanf("%f %f %f %f %f %f",
          &frac,&radix,&v, &Iex, &L, &total);

```

```

printf("frac = %f,radix=%f, v=%f, Iex = %f, L = %f0,
      frac, radix, v, Iex, L);

lnr = log(radix);
Q4 = pow(2., - frac); /* In theory this is Q0 */
QL = pow(2., - (frac - Iex + L));
printf("Q4 / 2 = %f, QL / 2 = %f0,Q4/2.,QL/2.);

star = ' ';
starl = 'o';

a1 = EL(v) - EL2(v) - Q4 / 2.;
c2 = EL(v) - EL1(v) + Q4 / 2.;

if (Q4 <= (EL2(v) - EL1(v))) {
    a2 = EL(v) - EL2(v) + Q4 / 2.;
    b2 = EL(v) - EL1(v) - Q4 / 2.;
    condition = 1;
    printf("***** condition = 1 *****0);
}

else if (Q4 > (EL2(v) - EL1(v))) {
    a2 = EL(v) - EL1(v) - Q4 / 2.;
    b2 = EL(v) - EL2(v) + Q4 / 2.;
    condition = 2;
    printf("***** condition = 2 *****0);
}

b1 = a2;
c1 = b2;
low = a1;
high = c2;
step = (high - low) / total;

eps = 1;
for (i = low; i <= high; i += step){
    if ((i >= a1) && (i < a2))
        density[eps] = ef1(v, i) * (1. / (Q4 * QL));
    else if ((i >= b1) && (i < b2) && (condition == 1))
        density[eps] = ef2(v, i) * (1. / (Q4 * QL));
    else if ((i >= b1) && (i < b2) && (condition == 2))
        density[eps] = 1. / Q4;
    else if ((i >= c1) && (i < c2))
        density[eps] = ef3(v, i) * (1. / (Q4 * QL));
    else
        printf(" density[%d] at %f = %e0,
              eps, i, density[eps]);

    if (density[eps] <= 0.0)
        density[eps] = 0.0;
    fprintf(fp,"%d%e0, eps, density[eps]);
    eps++;
}

```

```

fclose(fp);
temp = 0.;

for (eps = 1; eps <= total; eps++)
    if (density[eps] > temp)
        temp = density[eps];

if (temp >= 70.){
    temp = temp / 70.;
    for (eps = 1; eps <= total; eps++)
        density[eps] /= temp + 1.0;
}

for (eps = 1; eps <= total; eps++){
    printf("%d", eps);

    for (k = 1; k <= (density[eps]); k++)
        printf("%c", star);

    printf("%c0", star1);
}
}

double efl(delta, chi)
double delta, chi;
{
    extern double radix, lnr, QL;
    double Gammal(), res;
    res = - delta + QL/2. - log(Gammal(delta)/
        pow(radix, chi)-1.) / lnr;
    return(res);
}

double ef2(delta, chi)
double delta, chi;
{
    extern double radix, lnr;
    double Gammal(), res, Gamma2();
    res = (log(Gamma2(delta) / pow(radix, chi) - 1.) -
        log(Gammal(delta) / pow(radix, chi) - 1.)) / lnr;
    return(res);
}

double ef3(delta, chi)
double delta, chi;

```

```

{
    extern double radix, QL, lnr;
    double res, Gamma2();
    res = delta + QL / 2. + log(Gamma2(delta)
                               / pow(radix, chi) - 1.) / lnr;
    return(res);
}

double Gamma1(delta)
double delta;
{
    extern double radix, Q4;
    double res;
    res = (1. + pow(radix, - delta)) *
          pow(radix, - Q4 / 2.);
    return(res);
}

double Gamma2(delta)
double delta;
{
    extern double radix, Q4;
    double res;
    res = (1. + pow(radix, - delta)) * pow(radix, Q4 / 2.);
    return(res);
}

double EL1(delta)
double delta;
{
    extern double radix, lnr, QL;
    double res;
    res = log(1. + pow(radix, - (delta + QL / 2.))) / lnr;
    return(res);
}

double EL2(delta)
double delta;
{
    extern double radix, lnr, QL;
    double res;
    res = log(1. + pow(radix, - (delta - QL / 2.))) / lnr;
    return(res);
}

```

```
}  
  
double EL(delta)  
double delta;  
{  
  
    extern double radix;  
    double res;  
    res = log(1. + pow(radix, - delta)) / lnr;  
    return(res);  
}
```

APPENDIX I
CODE FOR GENERATION OF APPROXIMATION CURVE OF THE
p.d.f. OF THE ERROR AT THE OUTPUT OF THE LOGARITHMIC ADDER

```

/*****
*
* This program computes the approximated p.d.f.
* for the ERROR E at the output of the
* Logarithmic Adder for a specific address v.
* Accepts the base (radix), Fractional wordlength (frac)
* at the output of the memory table
* Iex is the number of integer (only) bits that the LNS
* exponent x is using.
* v determines the table address
* L is the number of shifts of the address performed
* It passes its output to the file 'outdenappr'
* to be used for plotting purposes.
*
*****/

#include <math.h>
#include <stdio.h>
#define SIZE 1000
double radix, lnr, Q4, QL;
main()
{
    extern double radix, lnr, Q4, QL;
    double temp, density[SIZE], a1, L, a2, b1, b2, c1, c2;
    double top, EL(), EL1(), EL2();
    char star, star1;
    int eps, k;
    double Iex, Prob1, Prob2, Prob3, total, high, frac;
    int condition;
    double sum1, sum2, dd1, dd2, v, sum3, delta1, delta2;
    double i, suma, sumb, sumc, j, dd3, l, m;
    double Imean(), step, low, Zero, Ivar(), Prob(), sum;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outdenappr", "w");

    printf(" frac, radix, v, Iex ,L, total0);
    scanf("%f %f %f %f %f %f",

```

```

&frac,&radix,&v, &Iex, &L, &total));
printf("frac=%f,radix=%f, v = %f, Iex = %f, L = %f0,
      frac, radix, v, Iex, L);

```

```

lnr = log(radix);
Q4 = pow(2., - frac);
QL = pow(2., - (frac - Iex + L));

printf("Q4 / 2 = %f, QL / 2 = %f0,
      Q4 / 2., QL / 2.);

```

```

star = ' ';
star1 = 'o';

```

```

top = EL2(v) - EL1(v);
a1 = EL(v) - EL2(v) - Q4 / 2.;
c2 = EL(v) - EL1(v) + Q4 / 2.;
if (Q4 <= top) {
    a2 = EL(v) - EL2(v) + Q4 / 2.;
    b2 = EL(v) - EL1(v) - Q4 / 2.;
    condition = 1;
    printf("***** condition = 1 *****0);
}
else if (Q4 > top) {
    a2 = EL(v) - EL1(v) - Q4 / 2.;
    b2 = EL(v) - EL2(v) + Q4 / 2.;
    condition = 2;
    printf("***** condition = 2 *****0);
}
b1 = a2;
c1 = b2;
low = a1;
high = c2;
step = (high - low) / total;

```

```

printf("a1 = %f a2 = %f0, a1, a2);
printf("b1 = %f b2 = %f0, b1, b2);
printf("c1 = %f c2 = %f0, c1, c2);

```

```

eps = 1;

```

```

if (condition == 1){

```

```

    for (i = low; i <= high; i += step){

```

```

        if ((i >= a1) && (i < a2)){
            density[eps] = (i - a1) / (top * Q4);
            printf("efl1 at %d = %e0,eps,density[eps]);
        }
        else if ((i >= b1) && (i < b2)) {
            density[eps] = 1. / top;
            printf("efl2 at %d = %e0,eps,density[eps]);
        }
    }
}

```

```

        else if ((i >= c1) && (i < c2))    {
            density[eps] = - (i - c2) / (top * Q4);
            printf("ef13 at %d = %e0,eps,density[eps]);
        }
        fprintf(fp, "%d%f0, eps, density[eps]);
        eps++;
    }

}
else if (condition == 2){
    for (i = low; i <= high; i += step){
        if ((i >= a1) && (i < a2))    {
            density[eps] = (i - a1) / (Q4 * top);
            printf("ef21 at %d = %e0,eps,density[eps]);
        }
        else if ((i >= b1) && (i < b2))    {
            density[eps] = 1. / Q4;
            printf("ef22 at %d = %e0,eps,density[eps]);
        }
        else if ((i >= c1) && (i < c2))    {
            density[eps] = - (i - c2) / (Q4 * top);
            printf("ef23 at %d = %e0,eps,density[eps]);
        }
        fprintf(fp, "%d%f0, eps, density[eps]);
        eps++;
    }

}

    fclose(fp);
    temp = 0.;

    for (eps = 1; eps <= total; eps++)
        if (density[eps] > temp)
            temp = density[eps];

    if (temp >= 70.){
        temp = temp / 70.;
        for (eps = 1; eps <= total; eps++)
            density[eps] /= temp + 1.0;
    }

    for (eps = 1; eps <= total; eps++){
        printf("%d", eps);
        for (k = 1; k <= (density[eps]); k++)
            printf("%c",star);
        printf("%c0,star1);
    }

}

```


APPENDIX J
SIMULATION OF A LOGARITHMIC ADDER DETERMINING
AN EXPERIMENTAL ERROR p.d.f.

```

/*****
*
* This program : 'Adder sim.c' performs a simulation
* for the Error analysis at the output of the
* Logarithmic Adder. Passes its output to the file
* 'outadder' to be used for plotting.
*
* The parameters are:
*
* Total is the number of random #s (10000);
* seed1 & seed2 are the random #s generator's
* initializers. The final result is the 'Histogram
* value divided by (TOTAL * step) where
* step = (high - low) / HISTO_AXIS,
* (Used in 'histo' function).
* frac are the fractional bits of the output and
* lex are the integer bits that ex is using.
* v is the # for which the histogram is performed.
*
*****/

```

```

#include <stdio.h>
#include <math.h>
#define SIZE 40000
int HISTO_AXIS;
double radix, lnr, Q4, QL;

main()
{
    extern int HISTO_AXIS;
    extern double radix, lnr, Q4, QL;
    double EL1(), EL2();
    int total;
    int i, low, high, le, L, seed1, seed2;
    double temp, K, v, low1, frac, high1;
    double E4[SIZE], EL[SIZE], E[SIZE];
    char nam[20];

```

```

short rand(), nfrom());

printf("Enter HISTO AXIS :0);
scanf("%d", &HISTO AXIS);
printf("Enter radix, total, frac, v, Ie, L :0);
scanf("%f %d %f %f %d %d",
      &radix, &total, &frac, &v, &Ie, &L);
printf("Enter seed1, seed2 :0);
scanf("%d %d", &seed1, &seed2);

lnr = log(radix);
K = log(1. + pow(radix, - v)) / lnr;
printf("radix = %f total = %d and frac = %f0,
radix, total, frac);
printf("v = %f Ie = %d K = %f and L = %d0,
      v, Ie, K, L);

Q4 = pow(2., - frac);
QL = pow(2., - (frac - Ie + L));

printf("Q4 = %e QL = %e0, Q4, QL);

(void) srand(seed1);
low = 0;
high = 30000;
for (i = 1; i <= total; i++)
    E4[i] = ((double)(nfrom(low,high)) *
            (Q4 / high)) - (Q4 / 2.);
(void) srand(seed2);
for (i = 1; i <= total; i++)
    EL[i] = ((double)(nfrom(low,high)) *
            (QL / high)) - (QL / 2.);

/* Histogram for the Output Error */
low1 = - Q4 / 2.;
high1 = Q4 / 2.;
histo(low1, high1, total, "E4", E4);

/* Histogram for the Error EL */
low1 = - QL / 2.;
high1 = QL / 2.;
histo(low1, high1, total, "EL", EL);

for (i = 1; i <= total; i++)
    E[i] = K - log(1. + pow(radix, -
        (v + EL[i]))) / lnr - E4[i];

/* Histogram for the final Error E */
low1 = K - EL2(v) - Q4 / 2.;
high1 = K - EL1(v) + Q4 / 2.;
histo(low1, high1, total, "E", E);

```

}

APPENDIX K
CODE USED TO GENERATE THE ENTRIES OF TABLE 6.1

```

/*****
*
*           Computation of approximation of
*           ERROR MEAN and VARIANCE
*           for various ranges of delta.
*
*   The computation is based on calculating the mean
*   and the variance of the output error
*   (due to quantization errors - and shifting -
*   at the input) for a specific delta, and then
*   averaging this variance by integrating the variance
*   over a range of delta's, and dividing the value of
*   the integral by the range length.
*
*   frac are the fractional bits of the output and
*   deltac0 determines the "essential zero"
*   Iex are the integer bits that x is using.
*
*   The experiment is performed for two and for
*   three levels of partitioning and no partitioning
*   at all.
*           program : Breakpoint.c
*
*****/

```

```

#include <math.h>
#include <stdio.h>
double lnr, Q4, QL, radix;
main()
{
    double Iex, delta3, sum3, delta1, Prob1, Prob2, frac;
    double sum1, sum2, dd1, dd2, deltac0, delta2;
    double i, suma, sumb, sumc, j, Prob3, dd3, l, m;
    double lmean(), Zero, lvar(), Prob(), sum;
    printf(" enter frac, radix, deltac0, Iex : 0);
    scanf("%f %f %f %f", &frac, &radix, &deltac0, &Iex);

    printf("  THREE-LEVEL EXPERIMENT  for :0);

```

```

printf("frac = %f, radix = %f, deltac0 = %f, Iex = %f0,
      frac, radix, deltac0, Iex);
printf("*****");
printf("*****0);
lnr = log(radix);
Q4 = pow(2., - frac);
/* In theory this is the precision */
Zero = pow(2., deltac0);
/* This is the essential zero */
/*
  Calculation of the average variance starts here:
*/
printf("Breakpointsvariance1variance2variance3");
printf("Total variance0);
i = 1.;
j = 1.;
delta3 = Q4;

for (j = 1.; j <= deltac0 + frac - 2.; j++){
  i = deltac0 - j;
  if (i < Iex - 2.) {
    delta1 = pow(2., i);
    ddl = Zero - delta1;
    sum1 = 0.;
    sum2 = 0.;
    sum3 = 0.;
    Prob1 = Prob(Zero, Zero, delta1);
    QL = pow(2., - frac + Iex); /* L = 0 */
    sum1 = Ivar(Zero, delta1) / ddl;
    sum1 *= Prob1;
    printf("%f", delta1);
    printf(" %e0, sum1);

    /* Calculation for the second interval now. */

    for (m = 1.; m <= i + frac - 1.; m++){
      if ((i + m) < Iex - 2.) {
        l = i - m;
        delta2 = pow(2., l);
        dd2 = delta1 - delta2;
        QL = pow(2., -(frac - Iex) - i); /* i = L */
        Prob2 = Prob(Zero, delta1, delta2);
        sumb = (Imean(delta1, delta2) / dd2) * Prob2;
        sum2 = Ivar(delta1, delta2) / dd2;
        sum2 *= Prob2;
        printf("%f", delta2);
        printf(" %e0, sum2);
        dd3 = delta2 - delta3;
        QL = pow(2., -(frac - Iex) - (i + m));
        /* i+m = L */
        Prob3 = Prob(Zero, delta2, delta3);
        sumc = (Imean(delta2, delta3) / dd3) * Prob3;
        sum3 = Ivar(delta2, delta3) / dd3;

```

```

        sum3 *= Prob3;
        printf("%e %e0
                ,sum3,sum1+sum2 + sum3);
    }
    else
        printf("0");
}
}

else
    printf("0");
}
printf("    TWO LEVEL EXPERIMENT    for:");
printf("0rac = %frac=%f deltac0=%f Iex = %f0,
        frac, radix, deltac0, Iex);
printf("*****");
printf("*****0");

/*
  Calculation of the average variance starts here:
*/

printf("Breakpoints  variance1  variance2");
printf("    Total variance0);

i = 1.;
j = 1.;
delta3 = Q4;
for (j = 1.; j <= deltac0 + frac - 1.; j++){
    i = deltac0 - j;
    if (i < Iex - 2.) {
        delta2 = pow(2., i);
        ddl = zero - delta2;
        dd2 = delta2 - delta3;
        sum1 = 0.;
        sum2 = 0.;
        QL = pow(2., - frac + Iex);    /* L = 0    */
        Prob1 = Prob(zero, zero, delta2);
        sum1 = (Ivar(zero,delta2) / dd1);
        sum1 *= Prob1;
        printf("%f %e 0, delta2, sum1);

    /*    Calculation for the second interval now.    */

    Prob2 = Prob(zero, delta2, delta3);
    QL = pow(2., -(frac - Iex) - i);
    sumb = (Imean(delta2,delta3) / dd2) * Prob2;
    sum2 = Ivar(delta2,delta3) / dd2;
    sum2 *= Prob2;
    printf("%f %e %e0,
            delta2, sum2, sum1 + sum2);
    }

```

```

        else
            printf("0");
    }
    printf(" NO BREAKPOINT EXPERIMENT      for:");
    printf(" frac=%f radix=%f deltac0=%f Iex=%f0,
           frac, radix, deltac0, Iex);
    printf("*****");
    printf("*****0");
    /*
       Calculation of the average variance starts here:
    */
    delta3 = Q4;
    QL = pow(2., - frac + Iex);    /* L = 0    */
    sum = 0.;
    suma = 0.;
    ddl = Zero - delta3;
    sum = (Ivar(Zero,delta3) / ddl);
    suma = (Imean(Zero,delta3) / ddl);
    printf("Average TOTAL variance = %e0, sum);
    printf("Average TOTAL mean = %e0, suma);
    printf("Breakpoints Average variance");
    printf("Comprehensive variance0);
    sum = 0.;
    for (j = 0.; j <= deltac0 + frac; j++){
        i = deltac0 - j;
        delta1 = pow(2., i);
        delta2 = pow(2., i - 1.);
        ddl = delta1 - delta2;
        sum1 = 0.;
        Probl = Prob(Zero, delta1, delta2);
        sum1 = (Ivar(delta1,delta2) / ddl);
        sum1 *= Probl;
        sum += sum1;
        printf(" %f %f%e%e0,
              delta1, delta2, sum1, sum);
    }
}

double Imean(x2, x1)
double x1, x2;
{
    double res, mean();
    res = ((x2 - x1) / 6.) * (mean(x2)
        + 4. * mean((x1 + x2) / 2.) + mean(x1));
    return(res);
}

double Ivar(x2, x1)
double x1, x2;
{

```

```

double res, var();
res = ((x2 - x1) / 6.) *
      (var(x2) + 4. * var((x1 + x2) / 2.) + var(x1));
return(res);
}

double var(delta)
double delta;
{
    extern double radix, lnr, Q4, QL;
    double top, res, a1, a2, b1, b2, c1, c2;
    double EL(), EL1(), EL2(), mean(), ef1(), ef2(), ef3();
    int condition;
    top = EL2(delta) - EL1(delta);
    a1 = EL(delta) - EL2(delta) - Q4 / 2.;
    c2 = EL(delta) - EL1(delta) + Q4 / 2.;
    if (Q4 <= top) {
        a2 = EL(delta) - EL2(delta) + Q4 / 2.;
        b2 = EL(delta) - EL1(delta) - Q4 / 2.;
        condition = 1;
    }
    else if (Q4 > top) {
        a2 = EL(delta) - EL1(delta) - Q4 / 2.;
        b2 = EL(delta) - EL2(delta) + Q4 / 2.;
        condition = 2;
    }
    b1 = a2;
    c1 = b2;
    if (condition == 1) {
        res = ((pow(a2, 4.) - pow(a1, 4.)) / 4. +
              (pow(a1, 4.) - pow(a2, 3.) * a1) / 3.) / Q4;
        res += (pow(b2, 3.) - pow(b1, 3.)) / 3.;
        res -= ((pow(c2, 4.) - pow(c1, 4.)) / 4. -
              (pow(c2, 4.) - pow(c1, 3.) * c2) / 3.) / Q4;
        res /= top;
        res -= pow(mean(delta), 2.);
    }
    else if (condition == 2) {
        res = ((pow(a2, 4.) - pow(a1, 4.)) / 4. +
              (pow(a1, 4.) - pow(a2, 3.) * a1) / 3.) / (Q4 *
              (EL2(delta) - EL1(delta)));
        res += (pow(b2, 3.) - pow(b1, 3.)) / (3. * Q4);
        res -= ((pow(c2, 4.) - pow(c1, 4.)) / 4. -
              (pow(c2, 4.) - pow(c1, 3.) * c2) / 3.)
              / (Q4 * (EL2(delta) - EL1(delta)));
        res -= pow(mean(delta), 2.);
    }
    return(res);
}
}

```

```

double mean(delta)
double delta;
{
    extern double radix, lnr, Q4, QL;
    double top, res, a1, a2, b1, b2, c1, c2;
    double EL(), EL1(), EL2();
    int condition;
    top = EL2(delta) - EL1(delta);
    a1 = EL(delta) - EL2(delta) - Q4 / 2.;
    c2 = EL(delta) - EL1(delta) + Q4 / 2.;
    if (Q4 <= top) {
        a2 = EL(delta) - EL2(delta) + Q4 / 2.;
        b2 = EL(delta) - EL1(delta) - Q4 / 2.;
        condition = 1;
    }
    else if (Q4 > top) {
        a2 = EL(delta) - EL1(delta) - Q4 / 2.;
        b2 = EL(delta) - EL2(delta) + Q4 / 2.;
        condition = 2;
    }
    b1 = a2;
    c1 = b2;
    if (condition == 1) {
        res = ((pow(a2, 3.) - pow(a1, 3.)) /
            3. + (pow(a1, 3.) - pow(a2, 2.) * a1) / 2.) / Q4;
        res += (pow(b2, 2.) - pow(b1, 2.)) / 2.;
        res -= ((pow(c2, 3.) - pow(c1, 3.)) / 3. -
            (pow(c2, 3.) - pow(c1, 2.) * c2) / 2.) / Q4;
        res /= top;
    }
    else if (condition == 2) {
        res = ((pow(a2, 3.) - pow(a1, 3.)) / 3. + (pow
            (a1, 3.) - pow(a2, 2.) * a1) / 2.) / (Q4 * top);
        res += (pow(b2, 2.) - pow(b1, 2.)) / (2. * Q4);
        res -= ((pow(c2, 3.) - pow(c1, 3.)) / 3. - (pow(c2,
            3.) - pow(c1, 2.) * c2) / 2.) / (Q4 * top);
    }
    return(res);
}

double Prob(zero, w2, w1)
double zero, w2, w1;
{
    double res;
    res = (2. / zero) * ((w2 - w1) - (pow(w2, 2.) -
        pow(w1, 2.)) / (2. * zero));
    return(res);
}

```


APPENDIX L
CODE FOR LATENCY OPTIMIZATION OF AMP LNS

```

/*****
*
*   This program : 'tree.c' provides for a study of the
*   associative memory LNS processor in a tree fashion.
*   For each of the #s between Low and High, generates a
*   binary representation without including the decimal
*   point. It then forms a tree to generate the output
*   from the input and gives information about the total
*   number of switches and leaf nodes of the tree.
*   Output to 'screen'.
*   The parameters are:
*   base : rad, Input precision : prec defined by frac1,
*   Output precision : prec2 defined by frac2,
*   Low and High define the range of the v variable.
*   N (= K for a complete tree) specifies the # of bits
*   for the final binary input representation,
*   and shift2 for the binary Output (mapping).
*
*****/

#include <math.h>
#include <stdio.h>
#define SIZE 20
#define BIG 65538

int old s, n1L, shift2, n1H, Rlow;
int K, N, w[SIZE], n[SIZE], s[SIZE], index, Rhigh;
int lim, index, Total switches, final nodes, count;
double frac1, frac2, N1[BIG], Nh[BIG], low, high;
double round(), rad, prec, prec2, logr(), v;
double highvm[BIG], lowvm[BIG], highvt[BIG], lowvt[BIG];

main()
{

    int map, L[SIZE], produce();
    printf("Enter frac1, frac2, low, high, rad 0);

```

```

scanf("%f %f %f %f %f",&frac1,&frac2, &low, &high, &rad);

prec = pow(2., - frac1);
prec2 = pow(2., - frac2);

if (round(log(round(low, prec)), prec2) >= 0.99999)
    shift2 = frac2 + 1;
else
    shift2 = frac2;

if (high >= 1.)
    N = frac1 + floor(log(floor(high) / log(2.)) + 1;
else
    N = floor(log(pow(prec, -1.) * high) / log(2.)) + 1;

lim = (BIG<(int)pow(2.,(double)shift2)) ?
        BIG : pow(2.,(double)shift2);

printf("0N = %dF = %d0, N, shift2);
printf("Enter K=N ?,Rlow,Rhigh=1 ?n1L=?n1H = ?0);
scanf("%d %d %d %d %d", &K, &Rlow, &Rhigh, &n1L, &n1H);

printf("0 = %dK = %dRlow = %dRhigh = %d0,
        N, K, Rlow, Rhigh);
printf("frac1 = %.1frac2 = %.1flim = %d0,
        frac1, frac2, lim);
printf("Input precision = %gOutput precision = %g0,
        prec, prec2);
printf("low = %fhigh = %frad = %.1f0,
        low, high, rad);
printf("n1L = %dn1H = %d0, n1L, n1H);

for (index = 0; index < lim; index++){
    Nl[index] = 0;
    Nh[index] = 0;
    lowvm[index] = 0;
    highvm[index] = 0;
    lowvt[index] = 0;
    highvt[index] = 0;
}

for (index = 0; index < SIZE; index++){
    L[index] = 0;
    n[index] = 0;
    s[index] = 0;
    w[index] = 0;
}

L[1] = N - K + 1;

if (K == 1) {
    n[1] = N;
    printf("    n[1] = %d0, n[1]);

```

```

    }
    else
        for (index = 1; index <= K; index++)
            produce(L, index);
}

double logr(arg)
double arg;

{
    return(log(1. + pow(rad, - arg)) / log(rad));
}

double round(arg, Q)
double arg, Q;

{
    return(floor(0.5 + arg / Q) * Q);
}

produce(L, index)
int L[SIZE];
{
    int nodes, j, condition, Total_nodes, m, flag;
    int sumn, Parts, t, intv1, intv2, intmap1, intmap2;
    double v1, v2, map1, map2, ln;

    for (w[index] = 1; w[index] <= L[index]; w[index]++){
        n[index] = w[index];
        L[index+1] = L[index] - w[index] + 1;

        if (index < K - 1) {
            produce(L, index + 1);
        }

        else {
            n[index + 1] = L[index + 1];
            condition = 1;

            for (j = 1; j <= K; j++)
                if ((n[j]>Rhigh) || (n[j]<Rlow)
                    || (n[1]<n1L) || (n[1]>n1H))
                    condition = 0;

            if (condition == 1){

```

```

Total_switches = 0;
Total_nodes = 0;
old_s = 1;
sumn = 0;
Nl[1] = low;
Nh[1] = high;

for (j = 1; j <= K; j++){

    for (m = 1; m <= old_s; m++) {
        lowvm[m] = Nl[m];
        highvm[m] = Nh[m];
    } /* for m */

    count = 0;
    sumn += n[j];
    final_nodes = 0;

    for (m = 1; m <= old_s; m++) {

        Parts = ceil((highvm[m] - lowvm[m]) /
            (pow(2., (double) (N - sumn)) * prec));
        flag = 0;

        for (t = 1; t <= Parts; t++) {

            lowvt[t] = lowvm[m] + (t - 1) * prec
                * pow(2., (double) (N - sumn));
            highvt[t] = lowvt[t] + prec *
                (pow(2., (double) (N - sumn)) - 1);

            if (lowvt[t] < high) {

                for (v=lowvt[t];v<=highvt[t]
                    -prec;v+=prec){

                    v1 = round(v, prec);
                    v2 = round(v + prec, prec);
                    intv1 = pow(prec, -1.) * v1;
                    intv2 = pow(prec, -1.) * v2;
                    intv1 <<= (32 - sumn);
                    intv2 <<= (32 - sumn);
                    map1 = round(logr(v1), prec2);
                    map2 = round(logr(v2), prec2);
                    intmap1=pow(prec2, -1.) * map1;
                    intmap2=pow(prec2, -1.) * map2;
                    intmap1 <<= (32 - shift2);
                    intmap2 <<= (32 - shift2);

                    if ((intmap1 ^ intmap2) != 0) {

                        count++;

```

```

        if (count >= lim){
            printf("count = %d > lim0,
                    count);
            exit();
        }

        Nl[count] = lowvt[t];
        Nh[count] = highvt[t];
        break;

    } /* if */

    else if ((v2 ==
        round(highvt[t], prec)) &&
        (flag != 1)) {

        final_nodes++;
        flag = 1;

    } /* else if */

    } /* for v */

    } /* if lowvt */

    } /* for t */

} /* for m */

old_s = count;
Total_switches += count;
Total_nodes += final_nodes;

if (j == K - 1)
    nodes = old_s;

if (old_s == 0){
    s[j] = 0;
    printf("n[%d]=%ds[%d]=%dswitches=%d"
        ,j, n[j], j, s[j], old_s);
    printf("# of final nodes = %d0,
        final_nodes);
    break;
} /* if old_s */
else {
    ln = log((double) old_s) / log(2.);

    if (old_s == 1)
        s[j] = 1;

    else if ((ln - floor(ln)) <=
        0.0000000000000001)
        s[j] = floor(ln);

```

```

        else if (ln - floor(ln) > 0.)
            s[j] = floor(ln) + 1;
    } /* else */

    printf("n[%d]=%ds[%d]=%dswitches=%d"
           , j, n[j], j, s[j], old_s);
    printf("# of final nodes = %d0,
           final_nodes);

} /* for j */

printf("0");
printf("Total # of switches = %d0,
       Total_switches);
printf("Total # of nodes = %d0,
       Total_nodes+nodes);

if ((n[1] == N - K + 1) || (n[1] == Rhigh))
    exit();

} /* if condition */

} /* else index */

} /* for w[] */

} /* produce() */

```

APPENDIX M
CODE USED TO GENERATE THE ENTRIES OF TABLE 7.3

```

/*****
*
* This program : 'am_proc.c' provides for a study of the *
* associative memory LNS processor. For each of the #s *
* between Low and High, generates a binary representation*
* without including the decimal point. It then *
* partitions the representation array of N elements into *
* K adjoint subarrays and forms the N[i] and S[i] *
* parameters for the associative memory Logarithmic *
* processor study. *
*
* Output to 'screen'. *
*
* The parameters are: *
*
* radix : rad, Input precision : pre defined by frac1, *
* Output precision : prec2 defined by frac2, *
* Low and High define the range of the v variable. *
* N specifies the # of bits for the final *
* binary input representation, and *
* shift2 for the Output (mapping) representation *
*
*****/

#include <math.h>
#include <stdio.h>
#define SIZE 20
#define BIG 50000

int K, N, w[SIZE], n[SIZE], s[SIZE], index, Rhigh;
int index, shift2, old s, Rlow, count;
double frac1, frac2, NI[BIG], Nh[BIG], low, high;
double round(), rad, prec, prec2, logr(), v;
double highvm[BIG], lowvm[BIG], highvt[BIG], lowvt[BIG];

main()
{
    int L[SIZE], map, produce();

    printf("Enter frac1, frac2, low, high, rad 0);

```

```

scanf("%f %f %f %f %f",&frac1,&frac2,&low, &high, &rad);

prec = pow(2., - frac1);
prec2 = pow(2., - frac2);
map = round(logr(round(low, prec)), prec2);
map *= pow(prec2, -1.);

if (map >= 1.)
    shift2 = frac2 + 1;
else
    shift2 = frac2;

if (high >= 1.)
    N = frac1 + floor(log(floor(high)) / log(2.)) + 1;
else
    N = floor(log(pow(prec, -1.) * high) / log(2.)) + 1;

printf("N = %dF = %d0, N, shift2);
printf("Enter K, Rlow, Rhigh0);
scanf("%d %d %d", &K, &Rlow, &Rhigh);
printf("N = %dK = %dRlow = %dRhigh = %d0,
        N,K, Rlow, Rhigh);
printf("frac1 = %.1ffrac2 = %.1f0, frac1, frac2);
printf("Input precision = %g Output precision = %g0,
        prec, prec2);
printf("low = %fhigh = %frad = %.1f0,
        low, high, rad);

for (index = 0; index < BIG; index++){
    Nl[index] = 0;
    Nh[index] = 0;
    lowvm[index] = 0;
    highvm[index] = 0;
}
for (index = 0; index < SIZE; index++){
    L[index] = 0;
    n[index] = 0;
    s[index] = 0;
    w[index] = 0;
}

L[1] = N - K + 1;

if (K == 1) {
    n[1] = N;
    printf("    n[1] = %d0, n[1]);
}
else
    for (index = 1; index <= K; index++)
        produce(L, index);
}

```



```

        lowvm[m] = Nl[m];
        highvm[m] = Nh[m];
    }

    count = 0;
    sumn += n[j];
    printf("sumn = %d0, sumn);
    /*
    */

    for (m = 1; m <= old_s; m++) {

        Parts = ceil((highvm[m] - lowvm[m])/
        (pow(2.,(double) (N-sumn)) * prec));
        /*
        */
        printf("Parts = %d0, Parts);
        /*
        */

        for (t = 1; t <= Parts; t++) {

            lowvt[t]=lowvm[m]+(t-1)*prec
            * pow(2., (double) (N-sumn));
            highvt[t] = lowvt[t] + prec *
            (pow(2.,(double) (N-sumn))-1);

            if (lowvt[t] < high) {
                for (v=lowvt[t];
                    v<=highvt[t]-prec;v +=prec){

                    v1 = round(v, prec);
                    v2 = round(v + prec, prec);
                    intv1 = pow(prec, -1.) * v1;
                    intv2 = pow(prec, -1.) * v2;
                    intv1 <<= (32 - sumn);
                    intv2 <<= (32 - sumn);
                    map1=round(logr(v1), prec2);
                    map2=round(logr(v2), prec2);
                    intmap1=pow(prec2,-1.)*map1;
                    intmap2=pow(prec2,-1.)*map2;
                    intmap1 <<= (32 - shift2);
                    intmap2 <<= (32 - shift2);

                    if((intmap1 ^ intmap2) != 0){

                        count++;

                        if (count >= BIG)
                            printf("count=%d > %d0,
                                count,BIG);

                        Nl[count] = lowvt[t];
                        Nh[count] = highvt[t];
                        break;

```

```

    }
    }
    }
    }
}

old_s = count;

if (old_s == 0){
    s[j] = 0;
printf("n[%d] = %ds[%d] = %dstates = %d0
        , j, n[j], j, s[j], old_s);
    break;
}
else {
    ln = log((double) old_s) / log(2.);

    if (old_s == 1)
        s[j] = 1;

    else if ((ln - floor(ln))
              <= 0.000000000000000001)
        s[j] = floor(ln);
    else if (ln - floor(ln) > 0.)
        s[j] = floor(ln) + 1;
}

printf("n[%d] = %ds[%d] = %dstates = %d0
        , j, n[j], j, s[j], old_s);

}

printf("0);

if (n[1] == N - K + 1)
    exit();

}

}

}

```

APPENDIX N
PROGRAM FOR SIMULATION OF THE LNS CORDIC PROCESSOR

```

/*****
*
* This program performs a simulation for the
* conventional CORDIC trigonometric processor
* and the equivalent LNS trigonometric engine.
* The LNS results are also compared to a FLP
* implementation with double precision (64 bits)
* as well as to a FXP implementation.
*
* The parameters involved are :
*   frac, Qcor, Qlog : # of bits for FXP, CORDIC, and
*   LNS realization respectively.
*   step : determines the number of intervals in which
*   the 90° interval is split, for computation
*   of angles and radii.
*   Cordic_iter : gives the # of CORDIC iterations
*   necessary to achieve maximum possible precision.
*   const : K of Cordic equations ((9.4) or (9.5))
*
* Output offers the square of the error of all three
* realizations when compared to the FLP one.
*
*****/

#include <math.h>
#include <stdio.h>
#define size 1001
#define SIZE 1001
#define COEFF 10
#define pe 3.141592654
double rad, lowexp;
int Underflow;

main()
{
    double lx[SIZE], el3, el5, el7;
    double div, logadd(), step, logmul();
    double QT, logang[SIZE], gonia, sigma;

```

```

double TFLP,TFXP,Tlog,TCOR,FXPang[SIZE],FXPrad[SIZE];
double TFLPrad,TFXPrad, corrad[SIZE], TCORrad;
double Cordic_Iter, Tlograd, const;
double trunc(), Qcor,Xcord[SIZE];
double ATP[SIZE], angle[SIZE], Xcord[SIZE];
double lang[SIZE], temp, lrad[SIZE], corangle[SIZE];
double flpang[SIZE], Qlog,flprad[SIZE];
double cartx[SIZE],ly[SIZE], carty[SIZE];
double ICC(), frac, Int, Quant, Quant1;
double round(), el9, elpe2, FCC();
int j, i, signadd(), sign(), sx[SIZE], signmul();
int sang[SIZE], sdiv, stemp, sy[SIZE], srad[SIZE];

printf(" Enter frac, step 0);
printf(" Enter Cordic_Iter, const, Qcor, Qlog : 0);
scanf("%f%f%f%f%f",
      &frac, &step,&Cordic_Iter,&const, &Qcor, &Qlog);

Quant = pow(2.0, frac);
QT = pow(2.0, Qcor);
Quant1 = pow(2.0, Qlog);
rad = 2.0;
Int = 4.0;
el3 = round(FCC(3.), Quant1);
el5 = round(FCC(5.), Quant1);
el7 = round(FCC(7.), Quant1);
el9 = round(FCC(9.), Quant1);
elpe2 = round(FCC(pe / 2.), Quant1);
lowexp = - pow(2., Int);

printf("lowexp = %f0,lowexp );

for (i = 3; i <= 19; i++) { /* CORDIC Coefficients */
    ATP[i] = atan(pow(2., -(i - 3.))) * 180. / pe;
}

Underflow = 0;

printf("i radius angle sin cosine0);

for (i=0; i < (90. / step); i++) {
    cartx[i] = cos(pe * i * step / 180.);
    carty[i] = sin(pe * i * step / 180.);
    flprad[i] = hypot(cartx[i], carty[i]);
    flpang[i] = atan2(carty[i], cartx[i]) * 180. / pe;
    if (round(cartx[i], Quant) != 0.0)
        sigma = round(round(carty[i],Quant)
            /round(cartx[i], Quant), Quant);
    else
        sigma = round(carty[i] / cartx[i], Quant);
    lx[i] = round(FCC(cartx[i]), Quant1);
    sx[i] = sign(cartx[i]);
    ly[i] = round(FCC(carty[i]), Quant1);

```

```

sy[i] = sign(carty[i]);
FXPrad[i] = round(sqrt(round
    (pow(round(carty[i],Quant),2.), Quant) +
    round(pow(round(cartx[i], Quant), 2.),
    Quant)), Quant);
lrad[i] = logadd(2. * lx[i], sx[i], 2. *
    ly[i], sy[i]) / 2.;
srad[i] = signadd(2. * lx[i], sx[i], 2. *
    ly[i], sy[i]);
lrad[i] = ICC(lrad[i], srad[i]);
div = ly[i] - lx[i];
sdiv = sy[i] * sx[i];

if (pow(sigma, 2.) < 1.) {

    FXPang[i] = round(round(round((round(sigma, Quant)
        - round(round(pow(sigma, 3.), Quant) / 3., Quant)
        + round(round(pow(sigma, 5.), Quant) / 5., Quant)
        - round(round(pow(sigma, 7.), Quant) / 7., Quant)
        + round(round(pow(sigma, 9.), Quant) / 9., Quant)
        ), Quant) * 180., Quant) / pe, Quant);
    lang[i] = logadd(div, sdiv, 3. * div - el3, - sdiv);
    sang[i] = signadd(div, sdiv, 3.
        * div - el3, - sdiv);
    temp = logadd(lang[i], sang[i], 5.
        * div - el5, sdiv);
    stemp = signadd(lang[i], sang[i], 5.
        * div - el5, sdiv);
    lang[i] = logadd(temp, stemp, 7.
        * div - el7, - sdiv);
    sang[i] = signadd(temp, stemp, 7.
        * div - el7, - sdiv);
    lang[i] = logadd(lang[i], sang[i], 9.
        * div - el9, sdiv);
    sang[i] = signadd(lang[i], sang[i], 9.
        * div - el9, sdiv);

}

else if (sigma >= 1.) {

    FXPang[i] = round(round(round((round(1./round(-sigma,
        Quant),Quant) + round(round(pe, Quant) / 2., Quant)
        + round(1./round(round(pow(sigma, 3.), Quant)
        * 3., Quant), Quant) - round(1./round(round(pow(
        sigma, 5.), Quant) * 5., Quant), Quant) + round(1.
        /round(round(pow(sigma, 7.), Quant) * 7., Quant),
        Quant) - round(1./round(round(pow(sigma, 9.),
        Quant) * 9., Quant), Quant)), Quant) * 180.,
        Quant) / pe, Quant);
    lang[i] = logadd(- div, - sdiv, - 3.
        * div - el3, sdiv);
    sang[i] = signadd(- div, - sdiv, - 3.

```

```

    * div - el3, sdiv);
temp = logadd(lang[i], sang[i], - 5.
    * div - el5, - sdiv);
stemp = signadd(lang[i], sang[i], - 5.
    * div - el5, - sdiv);
lang[i] = logadd(temp, stemp, - 7.
    * div - el7, sdiv);
sang[i] = signadd(temp, stemp, - 7.
    * div - el7, sdiv);
lang[i] = logadd(lang[i], sang[i], - 9.
    * div - el9, - sdiv);
sang[i] = signadd(lang[i], sang[i], -9.
    * div - el9, - sdiv);
lang[i] = logadd(lang[i], sang[i], elpe2, 1);
sang[i] = signadd(lang[i], sang[i], elpe2, 1);
}

else if (sigma <= - 1.) {
FXPang[i] = round(round(round((round(1./round(-sigma,
Quant),Quant) - round(round(pe, Quant) / 2., Quant)
+ round(1. /round(round(pow(sigma, 3.), Quant) * 3.,
Quant), Quant) - round(1. /round(round(pow(sigma, 5.
), Quant) * 5., Quant), Quant) + round(1. /round(
round(pow(sigma, 7.), Quant) * 7., Quant), Quant) -
round(1. /round(round(pow(sigma, 9.), Quant) * 9.,
Quant), Quant)), Quant) * 180., Quant) / pe, Quant);
lang[i] = logadd(- div, - sdiv, - 3.
    * div - el3, sdiv);
sang[i] = signadd(- div, - sdiv, - 3.
    * div - el3, sdiv);
temp = logadd(lang[i], sang[i], - 5.
    * div - el5, - sdiv);
stemp = signadd(lang[i], sang[i], - 5.
    * div - el5, - sdiv);
lang[i] = logadd(temp, stemp, - 7. * div - el7, sdiv);
sang[i] = signadd(temp, stemp, - 7. * div - el7, sdiv);
lang[i] = logadd(lang[i], sang[i], - 9.
    * div - el9, - sdiv);
sang[i] = signadd(lang[i], sang[i], -9.
    * div - el9, - sdiv);
lang[i] = logadd(lang[i], sang[i], elpe2, - 1);
sang[i] = signadd(lang[i], sang[i], elpe2, - 1);
}
logang[i] = ICC(lang[i], sang[i]) * 180. / pe;
logang[i] -= floor(logang[i] / 90.) * 90.;
/* CORDIC STARTS HERE */
Ycord[1] = trunc(carty[i], QT);
Xcord[1] = trunc(cartx[i], QT);
if (Ycord[1] == 0.)
    corangle[i] = 0.0;
else if (Ycord[1] > 0.) {
    Ycord[2] = - trunc(Xcord[1], QT);
    Xcord[2] = trunc(Ycord[1], QT);

```

```

        angle[2] = 90.;
    }
    else if (Ycord[1] < 0.) {
        Ycord[2] = trunc(Xcord[1], QT);
        Xcord[2] = - trunc(Ycord[1], QT);
        angle[2] = - 90.;
    }
    if (Ycord[2] == 0.)
        corangle[i] = 0.0;
    else {
        j = 3;
        while ((j <= Cordic_Iter) && (Ycord[j - 1] != 0.0)){
            Ycord[j] = trunc(Ycord[j - 1] - sign(Ycord[j - 1])
                * Xcord[j - 1] * pow(2., -(j - 3.)), QT);
            Xcord[j] = trunc(Xcord[j - 1] + sign(Ycord[j - 1])
                * Ycord[j - 1] * pow(2., -(j - 3.)), QT);
            angle[j] = trunc(angle[j-1] + sign(Ycord[j-1])
                * ATP[j], QT);
            j++;
        }
        corangle[i] = angle[j-1];
        corrad[i] = Xcord[j-1] / const;
    }
    printf("%d FLP%f%f%f0, i,
        flprad[i], flpang[i], cartx[i]);
    printf("%d FXP%f%f0, i, FXPrad[i], FXPang[i]);
    printf("%d log%f%f0, i,
        lrad[i], logang[i] - floor(logang[i] / 90.) * 90.);
    printf("%d COR%f%fCONST = %f0, i,
        corrad[i], corangle[i], Xcord[j-1]);
}
TFLPrad = 0.0;
TFLP = 0.0;
TFXPrad = 0.0;
TFXP = 0.0;
Tlograd = 0.0;
Tlog = 0.0;
TCORrad = 0.0;
TCOR = 0.0;
for (i = 1; i < (90. / step); i++) {
    if (fabs(logang[i] - 45.) > 7.) {
        TFLP += pow(flprad[i], 2.);
        TFXP += pow(FXPrad[i] - flprad[i], 2.);
        Tlog += pow(logang[i] - flpang[i], 2.);
        TCOR += pow(corangle[i] - flpang[i], 2.);
        TFLPrad += pow(flprad[i], 2.);
        TFXPrad += pow(FXPrad[i] - flprad[i], 2.);
        Tlograd += pow(lrad[i] - flprad[i], 2.);
        TCORrad += pow(corrad[i] - flprad[i], 2.);
    }
}
TFXP = sqrt(TFXP / TFLP);
Tlog = sqrt(Tlog / TFLP);

```



```

TCOR = sqrt(TCOR / TFLP);
TFXPrad = sqrt(TFXPrad / TFLPrad);
Tlograd = sqrt(Tlograd / TFLPrad);
TCORrad = sqrt(TCORrad / TFLPrad);
printf("*** ERRORS :0FXP = %fTlog = %fTCOR = %f0,
        TFXP, Tlog, TCOR);
printf("0FXPrad = %fTlograd = %fTCORrad = %f0,
        TFXPrad, Tlograd, TCORrad);
}

double round(arg, Q)
double arg, Q;
{
    double res;
    res = floor(0.5 + arg * Q) / Q;
    return(res);
}

double trunc(arg, Q)
double arg, Q;
{
    double res;
    res = floor(arg * Q) / Q;
    return(res);
}

double logadd(arg1, s1, arg2, s2)
/* Produces the LNS magnitude exponent of the
   addition of two numbers */
double arg1, arg2;
int s1, s2;
{
    double res, temp;
    double round(), addmap(), submap();
    /* Round the address v to 8 bits */
    temp = round(arg1 - arg2, pow(2., 8.));
    if (s1 == 0)
        res = arg2;
    else if (s2 == 0)
        res = arg1;
    else if (s2 == s1) {
        if (temp >= 0.)
            res = arg1 + addmap(temp);
        else
            res = arg2 + addmap(-temp);
    }
    else {
        if (temp > 0.)
            res = arg1 + submap(temp);

```

```

        else if (temp == 0.)
            res = 0.; /* sign is also set to 0 */
        else
            res = arg2 + submap( - temp);
    }
    return(res);
}

double addmap(arg)
/* Produces the unrounded logarithmic mapping corresponding
   to the addition of two numbers (given by  $\Phi(v)$ ) */
double arg;

{
    extern double rad;
    double res;
    res = log(1. + pow(rad, - arg)) / log(rad);
    return(res);
}

double submap(arg)
/* Produces the logarithmic mapping corresponding to the
   subtraction of two numbers (given by  $\Psi(v)$ ) */
double arg;

{
    extern double rad;
    double res;
    res = log(1. - pow(rad, - arg)) / log(rad);
    return(res);
}

signadd(arg1, s1, arg2, s2)
/* Determines the first sign of the LNS exponent of the
   result of the addition (subtraction) of two numbers */
double arg1, arg2;
int s1, s2;

{
    int sign;
    double temp;
    if (s1 == 0)
        sign = s2;
    else if (s2 == 0)
        sign = s1;
    else if (s2 == s1)
        sign = s1;
    else {
        temp = arg1 - arg2;
        if (temp > 0.)
            sign = s1;
        else if (temp == 0.)
            sign = 0;
    }
}

```

```

        else
            sign = s2;
    }
    return(sign);
}

sign(arg) /* Determines the
           first LNS sign (of the real number) */
double arg;

{
    int sign;
    if (arg >= pow(rad, lowexp))
        sign = 1;
    else if (fabs(arg) < pow(rad, lowexp))
        sign = 0;
    else
        sign = -1;
    return(sign);
}

double FCC(arg)
/* Returns the LNS exponent of the absolute value
   ( not rounded) */
double arg;

{
    extern double lowexp, rad;
    double res;
    if (fabs(arg) >= pow(rad, lowexp))
        res = log(fabs(arg)) / log(rad);
    else
        res = 0.0;
    return(res);
}

double ICC(arg,sign)
/* Performs the conversion from LNS to FLP */
double arg;
int sign;

{
    extern double rad;
    double res;
    res = pow(rad, arg) * sign;
    return(res);
}

```

APPENDIX O
CODE SIMULATING AND ANALYZING THE FLP, FXP AND LNS
VERSIONS OF ECHO CANCELLERS

```

/*****
*
* This program simulates an echo canceller implemented
* in the Floating Number System.
* Passes its output to the file 'outflp' .
*
* The parameters used are:
* TOTAL      : # of numbers ( x1 ... xN) <= SIZE = 1000
* Ensemble   : # of times that echo cancelling
*              is repeated.
* power of 2 : For power = 3 the normally distributed
*              numbers extend from -8 to 8.
* K, Alpha   : Loop gain and Alpha shown below
*              (0.0125, .3)
* Taps       : Number of weight coefficients
* Delay      :  $y[k] = A * x[k - \text{delay}]$ 
*
*****/

```

```

#include <math.h>
#include <stdio.h>
#define SIZE 1000
main()
{
    int t, i, k, delay, j, N, total, sum;
    int Ensemble, low, high;
    double var, mea, ave[SIZE];
    double limit, alpha, gauss_ran, mean, variance;
    double QF, K, Alpha;
    double y[SIZE], y_hat[SIZE];
    double x[SIZE], e[25][SIZE], h[256][SIZE], mse;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outflp", "w");
    printf(" TOTAL # of numbers (10000 ?),\n Ensemble size (3 ?) :0);
    scanf("%d%d", &total, &Ensemble);

```

```

printf("    Enter power of 2 for dynamic
        range(limit) (4 ?): 0);
scanf("%f", &limit);
printf("    Enter total # of TAPS (256 ?),
        DELAY (5 ?):0);
scanf("%d%d", &N, &delay);
low = 0;
high = 10000;
alpha = 6. / pow(2., limit);
sum = pow(pow(2., limit), 2.) / 3.;
mean = (alpha / 2.) * sum;
variance = (sum * pow(alpha, 2.)) / 12.;
printf(" Enter Loop gain, Alpha: 0);
scanf("%f%f", &K, &Alpha);
for (t = 1; t <= Ensemble; t++){
    mse = 0.;
    /* Generation of normally distributed numbers */
    for (i = 0; i < total + N; i++){
        x[i] = 0.;
        for (j = 0; j < total - 1; j++){
            gauss_ran = 0.;
            for (j = 1; j <= sum; j++){
                /* sum = 12 for : (sigma, a = 1) */
                gauss_ran += alpha * (double)
                    _nfrom(low,high)/(double) high);
                x[i] = gauss_ran - mean;
            }
        }
        for (i = 0; i <= N - 1; i++){
            for (k = 0; k <= total; k++){
                h[i][k] = 0;
            }
            /* LMS Algorithm for Echo cancellation */
            for (k = delay; k <= delay + total - 1; k++){
                y[k] = Alpha * x[k - delay];
                /* y[k]=A * x[k-delay] */
                y_hat[k] = 0.;
                for (i = 0; i <= N - 1; i++){
                    if (k >= i)
                        y_hat[k] += x[k - i] * h[i][k];
                }
                e[t][k] = y[k] - y_hat[k];
                for (i = 0; i <= N - 1; i++){
                    if (k >= i)
                        h[i][k+1] = h[i][k] + 2. * K*x[k-i]*e[t][k];
                    else
                        h[i][k+1] = h[i][k];
                }
                mse += pow(e[t][k], 2.);
            }
        }
        mse /= total;
        printf(" MSE = %f0, mse);
    }
}
for (k = delay; k <= delay + total - 1; k++){
    ave[k] = 0.;
}

```

```

    for (t = 1; t <= Ensemble; t++){
        ave[k] += pow(e[t][k], 2.);
    }
    ave[k] /= Ensemble;
    printf(" ave[%d] = %f0, k, ave[k]);
    fprintf(fp, "%d%f0, k - delay, ave[k]);
}
fclose(fp);
}

```

```

/*****
*
*                               FXP CANCELLER
*
* *****/

```

```

#include <math.h>
#include <stdio.h>
#define SIZE 1000
main()
{
    int t, i, k, delay, j, N, Qfix, total, sum;
    int Ensemble, low, high;
    double var, mea, rave[SIZE];
    double limit, alpha, gauss_ran, mean, variance;
    double QF, K, Alpha;
    double round(), trunc();
    double y[SIZE], e[25][SIZE], h[256][SIZE];
    double y_hat[SIZE], x[SIZE], roundmse;
    FILE *fp, *fopen(), *fclose();
    fp = fopen("outfxp", "w");
    printf(" TOTAL # of numbers (10000 ?),
           Ensemble size (3 ?) :0);
    scanf("%d%d", &total, &Ensemble);
    printf(" Enter power of 2
           for dynamic range(limit) (4 ?): 0);
    scanf("%f", &limit);
    printf(" Enter total # of TAPS (256 ?),
           DELAY (5 ?) :0);
    scanf("%d%d", &N, &delay);
    low = 0;
    high = 10000;
    alpha = 6. / pow(2., limit);
    sum = pow(pow(2., limit), 2.) / 3.;
    mean = (alpha / 2.) * sum;
    variance = (sum * pow(alpha, 2.)) / 12.;
    printf(" Enter Loop gain, Alpha, Qfix: 0);
    scanf("%f%f%d", &K, &Alpha, &Qfix);
    QF = pow(2., (double) (Qfix - limit - 1));
    /* 1 bit is for sign */
    for (t = 1; t <= Ensemble; t++){
        roundmse = 0.;

```

```

/* Generation of normally distributed numbers */
for (i = 0; i < total + N; i++){
    x[i] = 0.;
}
for (i = 0; i < total - 1; i++){
    gauss_ran = 0.;
    for (j = 1; j <= sum; j++){
        /* sum = 12 for : (sigma, a = 1) */
        gauss_ran += alpha * (double)
            (nfrom(low,high) / (double) high);
    }
    x[i] = gauss_ran - mean;
}
for (i = 0; i <= N - 1; i++){
    for (k = 0; k <= total; k++){
        h[i][k] = 0;
    }
    /* Echo canceller for fixed point version */
    for (k = delay; k <= delay + total - 1; k++){
        y[k] = round(round(Alpha,QF) *
            round(x[k - delay],QF),QF);
        /* y[k]=A * x[k-delay] */
        y_hat[k] = 0.;
        for (i = 0; i <= N - 1; i++){
            if (k >= i)
                y_hat[k] += round(round(x[k - i],QF) *
                    round(h[i][k],QF),35.);
            /* 35 are the bits which are available
               to the accumulator */
        }
        e[t][k] = y[k] - y_hat[k];
        for (i = 0; i <= N - 1; i++){
            if (k >= i)
                h[i][k+1] = round(round(h[i][k],QF) +
                    round(2. * K * round(x[k - i],QF) *
                        round(e[t][k],QF),QF),QF);
            else
                h[i][k+1] = round(h[i][k],QF);
        }
        roundmse += pow(e[t][k], 2.);
    }
    roundmse /= total;
    printf(" Fixed point MSE = %f0, roundmse);
}
for (k = delay; k <= delay + total - 1; k++){
    rave[k] = 0.;
    for (t = 1; t <= Ensemble; t++){
        rave[k] += pow(e[t][k], 2.);
    }
    rave[k] /= Ensemble;
    printf(" rave[%d] = %f0, k, rave[k]);
    fprintf(fp,"%d%f0, k - delay, rave[k]);
}
fclose(fp);
}

```

```

/*****
*
*                               LNS CANCELLER
*
*****/

```

```

#include <math.h>
#include <stdio.h>
#define SIZE 1500
double rad, lowexp;
int Underflow;
main()
{
    extern int Underflow;
    double var, mea;
    double limit, alpha, gauss_ran, mean, variance;
    double K, Alpha, LAlpha, logadd(), logmul(), QL, QF;
    double round(), lx[SIZE], le[SIZE], trunc();
    double y[SIZE], y_hat[SIZE], h[50][SIZE];
    double ly[SIZE], ly_hat[SIZE], lh[50][SIZE];
    double x[SIZE], e[101][SIZE], FCC(), Int, ICC();
    double ltemp, lave[SIZE], l2K, logmse;
    int sy[SIZE], se[SIZE], sh[50][SIZE];
    int stemp, s2K, Ensemble, SALpha, sign(), signmul();
    int sy_hat[SIZE], sx[SIZE], Qfix, Qlog, signadd();
    int t, i, k, delay, j, N, total, sum, low, high;
    FILE *fp, *fopen(), *fclose();
    fp= fopen("outlog1", "w");
    rad = 2.;
    printf(" TOTAL # of numbers (10000 ?),
        Ensemble size (3 ?):0);
    scanf("%d%d", &total, &Ensemble);
    printf("Enter power of 2 for
        dynamic range(limit) (4 ?) 0);
    printf(" and # of INT bits for
        logarithmic system (9 ?): 0);
    scanf("%f%f", &limit, &Int);
    lowexp = - pow(2., Int);
    printf(" Enter total # of TAPS (50 ?),
        DELAY (5 ?) :0);
    scanf("%d%d", &N, &delay);
    printf(" Enter Loop gain, Alpha, Qfix, Qlog : 0);
    scanf("%f%f%d%d", &K, &Alpha, &Qfix, &Qlog);
    QF = pow(2., (double) (Qfix - limit - 1));
    /* 1 bit is for sign */
    QL = pow(2., (double) (Qlog - 2 - Int));
    /* 2 bits are for signs */
    low = 0;
    high = 10000;
    alpha = 6. / pow(2., limit);
    sum = pow(pow(2., limit), 2.) / 3.;

```



```

mean = (alpha / 2.) * sum;
variance = (sum * pow(alpha, 2.)) / 12.;
Underflow = 0;
LAlpha = round(FCC(Alpha), 12.);
SAlpha = sign(Alpha);
12K = round(logmul(round(FCC(2.),12.), sign(2.),
    round(FCC(K),12.), sign(K)), QL);
s2K = signmul(round(FCC(2.),12.), sign(2.),
    round(FCC(K),12.), sign(K));
for (t = 1; t <= Ensemble; t++){
    logmse = 0.;
    /* Generation of normally distributed numbers */
    for (i = 0; i < total + N; i++){
        lx[i] = 0.;
        sx[i] = 0;
    }
    for (i = 0; i < total - 1; i++){
        gauss_ran = 0.;
        for (j = 1; j <= sum; j++)
            /* sum = 12 for : (sigma, a = 1) */
            gauss_ran += alpha *
                (double)(nfrom(low,high) / (double) high);
        x[i] = gauss_ran - mean;
        lx[i] = round(FCC(x[i]), 12.);
        sx[i] = sign(x[i]);
    }
    for (i = 0; i <= N - 1; i++){
        for (k = 0; k <= total; k++){
            lh[i][k] = 0.;
            sh[i][k] = 0;
        }
    }
    /* Echo canceller for Logarithmic version */
    for (k = delay; k <= delay + total - 1; k++){
        ly[k] = round(logmul(LAlpha, SAlpha,
            lx[k - delay], sx[k - delay]), QL);
        sy[k] = signmul(LAlpha, SAlpha,
            lx[k - delay], sx[k - delay]);
        /* y[k]=A * x[k-delay] */
        ly_hat[k] = 0.;
        sy_hat[k] = 0;
        for (i = 0; i <= N - 1; i++){
            if (k >= i){
                ltemp = round(logmul(lx[k - i], sx[k - i],
                    lh[i][k], sh[i][k]), QL);
                stemp = signmul(lx[k - i], sx[k - i],
                    lh[i][k], sh[i][k]);
                ly_hat[k] = round(logadd(ly_hat[k], sy_hat[k],
                    ltemp, stemp), QL);
                sy_hat[k] = signadd(ly_hat[k],
                    sy_hat[k], ltemp, stemp);
            }
        }
    }
}

```

```

le[k] = round(logadd(ly[k], sy[k],
                    ly_hat[k], - sy_hat[k]), QL);
se[k] = signadd(ly[k], sy[k],
               ly_hat[k], - sy_hat[k]);
for (i = 0; i <= N - 1; i++){
    if (k >= i){
        stemp = signmul(lx[k - i],
                        sx[k - i], le[k], se[k]);
        ltemp = round(logmul(lx[k - i], sx[k - i],
                             le[k], se[k]), QL);
        stemp = signmul(l2K, s2K, ltemp, stemp);
        ltemp = round(logmul(l2K, s2K,
                             ltemp, stemp), QL);
        lh[i][k+1] = round(logadd(lh[i][k], sh[i][k],
                                   ltemp, stemp), QL);
        sh[i][k+1] = signadd(lh[i][k], sh[i][k],
                              ltemp, stemp);
    }
    else{
        lh[i][k+1] = lh[i][k];
        sh[i][k+1] = sh[i][k];
    }
}
e[t][k] = ICC(le[k], se[k]);
logmse += pow(e[t][k], 2.);
}
logmse /= total;
printf("logarithmic MSE = %f0, logmse);
}
for (k = delay; k <= delay + total - 1; k++){
    lave[k] = 0.;
    for (t = 1; t <= Ensemble; t++){
        lave[k] += pow(e[t][k], 2.);
    }
    lave[k] /= Ensemble;
    fprintf(fp, "%d%f0, k - delay, lave[k]);
}
fclose(fp);
printf(" Underflow = %d0, Underflow);
}

double logmul(arg1, s1, arg2, s2)
double arg1, arg2;
int s1, s2;
{
    double res;
    extern int Underflow;
    extern double lowexp;
    if ((s1 == 0) || (s2 == 0))
        res = 0.;
    else {

```

```

        res = arg1 + arg2;
        if (fabs(res) < pow(rad, lowexp))    {
            res = 0.;
            Underflow++;
        }
    }
    return(res);
}
signmul(arg1, s1, arg2, s2)
double arg1, arg2;
int s1, s2;
{
    extern double lowexp;
    int sign;
    if ((s1 == 0) || (s2 == 0))
        sign = 0;
    else if (fabs(arg1 + arg2) < pow(rad, lowexp))
        sign = 0;
    else if (s1 == s2)
        sign = 1;
    else
        sign = -1;
    return(sign);
}

```

BIOGRAPHICAL SKETCH

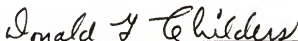
Thanos Stouraitis was born in Athens, Greece, on May 10, 1956. He earned a Bachelor of Science degree in physics and a master's degree in electronic automation from the University of Athens, Greece, in 1979 and 1981 respectively and a Master of Science in electrical engineering from the University of Cincinnati in 1983. For the period 1981 - 1983 he has been a researcher for NEMA labs. In August 1983 he entered the Graduate School of the University of Florida, where he is still serving as a graduate research assistant. His current research interests include very high speed digital computer architectures, and digital signal processing. He is coauthor of a monograph on design of digital filters (including CAD software for personal computers) and author of several technical papers. He is scheduled to receive his Doctor of Philosophy degree in August 1986.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Fred J. Taylor, Chairman
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



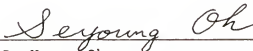
Donald G. Childers
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Julius T. Tou
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Se-Young Oh
Assistant Professor of
Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



George Logothetis
Assistant Professor of
Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 1986



Dean, College of Engineering

Dean, Graduate School